CS120: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Lecture 11: Graph Search

Harvard SEAS - Fall 2021 *Oct. 7, 2021*

1 Announcements

Recommended Reading:

- Roughgarden II Sec 8.1–8.2
- CLRS 22.2
- Salil OH after class today
- Midterm review posted
- Participation Portfolio 1 due Sun

2 Shortest Paths: First Attempts

Motivated by a (simplified version) of the Google Maps problem, we wish to design an algorithm for the following computational problem:

Input	: A directed graph $G = (V, E)$ and two vertices $s, t \in V$
Output	: A *shortest path* from s to t in G, or $\perp$ if no path from s to t exist

Computational Problem ShortestPaths

Definition 2.1. Let $G = (V, E)$ be a directed graph, and $s, t \in V$.

- A *path* p from s to t in G is a sequence $v_0, v_1, \ldots, v_\ell$ of vertices such that $v_0 = s$, $v_\ell = t$, and $(v_{i-1}, v_i) \in E$ for $i = 1, \ldots, \ell$.
- The *length* of a path p is length(p) = the number of edges in p (the number ℓ above).
- The *distance* from s to t in G is

$$\text{dist}_G(s,t) = \begin{cases} \min\{\text{length}(p) : p \text{ is a path from } s \text{ to } t\} & \text{if a path exists} \\ \infty & \text{otherwise} \end{cases}$$

- A *shortest path* from s to t in G is a path p from s to t with length$(p) = \text{dist}_G(s,t)$

Q: An algorithm immediate from the definition?
Enumerate over all paths from s in order of length, and terminate after finding the first that ends at t.

But when can we stop this algorithm if no path has been found? The following lemma allows us to stop at paths of length $n - 1$.

Lemma 2.2. *If p is a shortest path from s to t, then all of the vertices that occur on p are distinct.*

Proof.
Suppose for contradiction that there is a shortest path $p = (s = v_0, v_1, \ldots, v_l = t)$ that does *not* satisfy this property, i.e. $v_i = v_j$ for some $i < j$. But then we can cut out the loop $(v_i, v_{i+1}, \ldots, v_j)$ and produce the path $p' = (s = v_0, \ldots, v_{i-1}, v_i = v_j, v_{j+1}, \ldots, v_l)$. We have the length of p' is strictly less than that of p and has the same start and endpoints. But then p is not a shortest path, so we have a contradiction. □

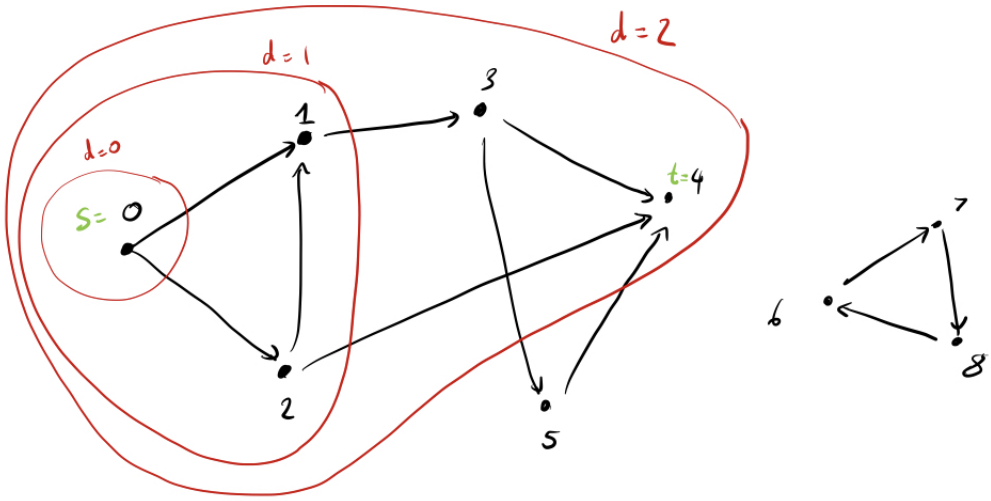
We can get a faster algorithm using *breadth-first search (BFS)*. For simplicity we'll start by presenting the algorithm for the following simpler computational problem:

Input	: A directed graph $G = (V, E)$ and two vertices $s, t \in V$
Output	: The distance from s to t in G

Computational Problem DistanceInGraph

```
1 BFSv0(G, s, t)
  Input    : A directed graph G = (V, E) and two vertices s, t ∈ V
  Output   : The distance from s to t in G
2 S = {s};
3 /* loop invariant:  S contains the vertices at distance ≤ d from s        */
4 foreach d = 0, ..., n − 1 do
5 |   if t ∈ S then return d;
6 |   S = S ∪ {v ∈ V : ∃u ∈ S s.t. (u, v) ∈ E}
7 return ∞
```

Example:
Consider the following graph $V = [9]$, $E = \{(0,1), (0,2), (1,3), (2,0), (2,1)$.



Q: What is happening at every iteration of the loop

We have a set of S which is the set of vertices that have been visited previously. At each iteration, we construct a new S' that is the union of S and the set of vertices that can be visited from all the vertices in S by one additional edge. This allows us to include the new vertices that can be visited now that we update the distance d.

Q: How do we prove correctness?

Establish the loop invariant from start of iteration d.

$$S = \{v \in V : dist_G(s,t) \leq d\}$$

We now can prove that the loop invariant holds by induction on d.

Base Case: At $d = 0$, $S = \{s\}$.

Induction Step: If $dist_G(s,v) = d$ then for $d > 0$ $\exists u$ s.t. $dist_G(s,u) = d-1$ and $(u,v) \in E$. Thus, we add everything at distance d. Conversely, if $dist_G(s,u) \leq d-1$ and $(u,v) \in E$, then $dist_G(s,v) \leq d$, so we did not add any extra vertices.

Then the loop invariant establishes that if the shortest path from s to t is of length k, we will add t to to S at exactly the kth iteration.

Q: What is the runtime of the algorithm? We have n iterations, and in each iteration, we perform the update of Line 6 in time $O(m)$ by enumerating over all possible edges (u,v) in E. (In order to be able to check whether $u \in S$ and add possibly add v to S in constant time, we can maintain S as a bitvector, i.e. an array of n bits, where the u'th entry is 1 iff $u \in S$.) This gives a total runtime $O(nm)$.

3 Improving BFS

Observations:

- S only grows due to edges that cross the *frontier* from S to $V - S$.
- Every edge in E crosses the frontier in at most one loop iteration.

```
1  BFS(G, s, t)
   Input  : A directed graph G = (V, E) and two vertices s, t ∈ V (in adjacency list
            representation)
   Output : The distance from s to t in G
2  S = {s};
3  F = {s} ;                                          /* the frontier vertices */
4  d = 0;
5  /* loop invariant:  S = vertices at distance ≤ d from s, F = vertices at
       distance d from s                                                        */
6  while F ≠ ∅ do
7  |   if t ∈ F then return d;
8  |   F = {v ∈ V − S : ∃u ∈ F s.t. (u, v) ∈ E};
9  |   S = S ∪ F;
10 |   d = d + 1;
11 return ∞
```

Theorem 3.1. `BFS(`G`)` *correctly solves DistanceInGraph and can be implemented in time* $O(n+m)$*, where* n *is the number of vertices in* G *and* m *is the number of edges.*

Proof. 1. Correctness:
Proof is similar to the prior argument.

2. Runtime:
To carry out the update in Line 8, we can enumerate over every *vertex* u in the frontier F, and try every edge (u, v) leaving that vertex and check if v lies in S. If we maintain S as a bitvector and maintain the frontier F as a linked list (e.g. a queue of vertices), then this will take time:

$$O\left(\sum_{u \in F}(1 + d_{out}(u))\right)$$

Then when we sum over all iterations of the loop, we use that each vertex only appears in at most one frontier, i.e. if we let F_d be the frontier at the d'th iteration, then the sets F_d are all disjoint. Thus, our total runtime is

$$O\left(\sum_{d=0}^{\infty}\sum_{u \in F_d}(1 + d_{out}(u)))\right) \leq O\left(\sum_{u \in V}(1 + d_{out}(u))\right) = O(n + m).$$

□

Above we used the following definition:

Definition 3.2. For a digraph $G = (V, E)$ and a vertex v, we define the *out-degree* of v to be

$$d_{out}(v) = |\{w : (v, w) \in E\}|$$

and the *in-degree* of v to be

$$d_{in}(v) = |\{u : (u, v) \in E\}|.$$

For an undirected graph, we have $d_{out}(v) = d_{in}(v)$, so we just call this the *degree* of v, denoted $d(v)$.