CS120: Intro. to Algorithms and their Limitations Anshu & Hesterberg

Lecture 10: Graph Search

Harvard SEAS - Fall 2023 *2023-10-05*

1 Announcements

- Please fill out the Problem Set 3 Survey. This is the best way to communicate changes you'd like to see for future weeks (lecture, problem sets, office hours, section).
- Reminder: Pset 3 solutions posted on Canvas
- Pset 4 released
- Sender-Receiver exercise on Tuesday.

Recommended Reading:

- Roughgarden II Sec 8.1–8.2
- CLRS 22.2

2 Graph Algorithms

Recommended Reading:

- Roughgarden II Sec 7.0–7.3, 8.0–8.1.1
- CLRS Appendix B.4

Motivating Problem: Google Maps. Given a road network, a starting point, and a destination, what is the shortest way to get from the starting point s to the destination t?

Q: How to model a road network?

3 Shortest Walks

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

Input	**:** A digraph $G = (V, E)$ and two vertices $s, t \in V$
Output	**:** A *shortest walk* from s to t in G, if any walk from s to t exists

Computational Problem ShortestWalk

Definition 3.1. Let $G = (V, E)$ be a directed graph, and $s, t \in V$.

- A *walk* w 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 walk w is $\mathrm{length}(w) =$ the number of edges in w (the number ℓ above).
- The *distance* from s to t in G is

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

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

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

But when can we stop this algorithm to conclude that there is no walk? The following lemma allows us to stop at walks of length $n - 1$.

Lemma 3.2. *If w is a shortest walk from s to t, then all of the vertices that occur on w are distinct.*

Proof.

□

A walk in which all vertices are distinct is also called a *path*.

Q: With this lemma, what is the runtime of exhaustive search?
A:

4 Breadth-First Search

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 $s, t \in V$
  Output : The distance from s to t in G
2 $S = \{s\}$;
3 `/* loop invariant:    */`
4 **foreach** $d = 0, \ldots, n-1$ **do**
5   **if** $t \in S$ **then return** d;
6   $S = S \cup \{v \in V : \exists u \in S \text{ s.t. } (u, v) \in E\}$
7 **return** ∞

Example:

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 perform the update of Line 6?

We'll iterate over all edges of G. We assume we are given the graph as an *adjacency list*: for each vertex v, we keep a neighbor array $\texttt{Nbr}[v] = \{u : (v, u) \in E\}$ holding the neighbors of v. We are also given the length of each such array $\texttt{Nbr}[v]$—we could compute these lengths ourselves, but they're so often useful that we'll save time by assuming the representation of the graph comes with them. [1] So we iterate over all edges of G by iterating over all those lists.

[1] Other ways of representing a graph are sometimes useful, and discussed in classes like CS 124. In CS 120, we'll always represent graphs by adjacency lists.

Q: How do we prove correctness?

Q: What is the runtime of the algorithm, in terms of the number of vertices n and the number of edges m?

5 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.

```
BFS(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
S = {s};
F = {s} ;                                   /* the frontier vertices */
d = 0;
/* loop invariant:    */
while F ≠ ∅ do
    if t ∈ F then return d;
    F = {v ∈ V − S : ∃u ∈ F s.t. (u, v) ∈ E};
    S = S ∪ F;
    d = d + 1;
return ∞
```

Theorem 5.1. 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:

2. Runtime:

□

Above we used the following definition:

Definition 5.2. 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 $d(v)$.

Q: How would we calculate the out-degree of v from the adjacency-list representation of a graph?

6 More Graph Search

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

Note that, by Lemma 3.2, shortest walks are paths, so we can use the terms "shortest paths" and "shortest walks" interchangeably.

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 $\text{dist}_G(s, v)$ (implicitly represented through a predecessor array as above)

Computational Problem SingleSourceShortestPaths

We have proven:

Theorem 6.1. *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.

7 (Optional) 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 do, however, briefly mention 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 walk 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 ∈ V, and a walk-length ℓ
Output : YES or NO
v = s;
foreach i = 1, ..., ℓ 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?

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 7.1. *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.*