CS1200: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Lecture 11: Graph Search

Harvard SEAS - Fall 2025 *2025-10-07*

1 Announcements

- Salil's OH Thu 1-1:45pm SEC 3.327.
- SRE 3 this Thursday: come prepared!
- Midterm Thu 10/16 (closed book). Practice midterms from 2022 and 2024 posted. This week's section will do midterm review!

Recommended Reading:

- Hesterberg–Vadhan Ch. 12
- CLRS 22.2
- Roughgarden II, 8.1–8.2
- Lewis & Zax 13, 16, 17

Where are we in the course trajectory?

2 Graphs

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?

A:

Definition 2.1.

Q: What possibilities doesn't this model capture and how might we augment it?

Unless we state otherwise, assume *graph* means a **simple, unweighted, undirected** graph, and a *digraph* means a **simple, unweighted, directed** graph. Graphs are very useful for modelling many kinds of relationships!

Representing Graphs.

Example 2.2. Consider the graph with vertices $V = [9]$ and edges

$$E = \{(0,1),(0,2),(1,3),(2,1),(2,4),(3,4),(3,5),(4,5),(6,7),(7,8),(8,6)\}.$$

- Adjacency list representation: For every vertex v, given
  - $\texttt{Nbr}_{\texttt{out}}[v] =$
  - $\deg_{out}(v) =$

  In above example, $\texttt{Nbr}_{\texttt{out}} =$
- Using Word-RAM with word length $w \geq \log n$, so vertex name fits in a single word.

3 Shortest Walks

Abstracting a simplified version of the route-finding problem above, 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 Shortest Walk

Let us define precisely what we mean by a *shortest walk*.

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 walk in which all vertices are distinct is also called a *path*.
- The *length* of a walk w is $\text{length}(w) =$
- The *distance* from s to t in G is
  $\text{dist}_G(s, t) =$
- A *shortest walk* from s to t in G is a walk w from s to t with $\text{length}(w) = \text{dist}_G(s, t)$

Q: What algorithm for ShortestWalk is immediate from the definition?

A:

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 (i.e. w is a* path*).*

Proof.

□

Because of this lemma, the Shortest Walk problem is usually referred to as the Shortest Path problem.

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

A:

4 Breadth-First Search

> "I don't know where I'm going, but I'm on my way." — George Fairman

We can get a much faster algorithm for Shortest Walk using *breadth-first search (BFS)*. For simplicity, we'll start by presenting algorithms to only compute the *length* of the shortest path from s to t, rather than actually find the path. On the other hand, our algorithm will actually compute the distance from s to *all* vertices in the graph, not only t. Let's capture these two modifications in the following definition:

Input: A digraph $G = (V, E)$ and a source vertex $s \in V$

Output: The array $\texttt{dist}_s$ where for every $t \in V$, $\text{dist}_s[t] = \text{dist}_G(s, t)$

Computational Problem Single-Source Distances

With this, here is our first version of BFS.

BFS(G, s):
Input : A digraph $G = (V, E)$ and a source vertex $s \in V$
Output : The array $\texttt{dist}_s[\cdot] = \text{dist}_G(s, \cdot)$
Initialize $\texttt{dist}_s[t] = \infty$ for all $t \in V$.;
$S = F = \{s\}$;
$\texttt{dist}_s[s] = 0$;
foreach $d = 1, \ldots, n-1$ **do**
  Let $F =$ ______________________________________ ;
  For every $v \in F$, $\texttt{dist}_s[v] = d$;
  $S = S \cup F$;
return dist_s

Algorithm 4.1: BFS for Single-Source Distances

Let's illustrate the behavior of BFS to the graph from Example 2.2 with $s = 0, t = 4$:

Q: What is happening at every iteration of the loop?

A:

Q: How do we prove correctness?

A:

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

A:

Q: How can we implement BFS faster?

A:

Putting all the above together, we obtain:

Theorem 4.1. $\texttt{BFS}(G)$ *correctly solves* SINGLE-SOURCE DISTANCES *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 in* G*.*

Implementation details.

- In practice, F is stored as a queue and updates are done one vertex u at a time rather than as a 'batch'.
- The bitvector S is redundant given that we are maintaining $\texttt{dist}_s$.

5 Finding the Paths

Q: How to actually find shortest *path* from s to t, not just the distances?

A:

Input: A digraph $G = (V, E)$ and a source vertex $s \in V$

Queries: For any query vertex t, return a shortest path from s to t (if one exists).

Abstract Data Type SINGLE-SOURCE SHORTEST PATHS

Theorem 5.1. SINGLE-SOURCE SHORTEST PATHS *has a data structure that, for digraphs with n vertices and m edges, has preprocessing time $O(n+m)$ and the time to answer a query t is $O(\mathrm{dist}_G(s,t))$.*

Given this, we deduce an algorithm for the SHORTEST WALK problem we started with, by running the preprocessor and then querying the data structure once with our given destination vertex t:

Corollary 5.2. SHORTEST WALK *(equivalently,* SHORTEST PATH*) can be solved in time $O(n + m)$.*

It is natural and very useful (e.g. in Google maps) to have data structures for variants of SHORTEST PATH on *dynamic graphs*. There is a rich collection of methods for dynamic graph data structures, which are beyond the scope of this course.

Another extension of BFS is to handle *weighted graphs*. This is Djikstra's algorithm, and is covered in CS 1240.

6 (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 this textbook, but DFS and some of its applications are covered in CS1240.

We do, however, briefly mention a randomized form of graph search, namely *random walks*, and use it to solve the *decision* problem of whether there *exists* a walk from s to t 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 UNDIRECTED S-T CONNECTIVITY

This problem can be solved by Algorithm 6.1, for an appropriate setting of the walk length ℓ.

```
RandomWalk(G, s, t, 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(deg_out(v));
    v = Nbr_out[v][j];
return ∞
```

Algorithm 6.1: `RandomWalk()`

Q: What is the advantage of this algorithm over BFS?

A:

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, t, ℓ) will visit all vertices reachable from s. Thus, we obtain:

Theorem 6.1. Undirected S-T Connectivity *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 (which is not modified during the algorithm).*