CS1200: Intro. to Algorithms and their Limitations	Anshu & Vadhan

Lecture 10: Graph Search

Harvard SEAS - Fall 2024 — *2024-10-03*

1 Announcements

- Sender-Receiver exercise on Tuesday.

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?

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.

Although we've started with this motivation of route-finding on road networks, in the coming lectures we'll see that graphs are useful for modelling a vast range of different kinds of relationships, e.g. social networks, the world wide web, kidney donor compatibilities, scheduling conflicts, etc. 50

Representing Graphs.

- Adjacency list representation: For every vertex v, given
  - $\texttt{Nbr}_{\texttt{out}}[v] =$
  - $\deg_{out}(v) =$
- Using Word-RAM with word length $w = O(\log n)$, 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 ShortestWalk

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 ShortestWalk problem is usually referred to as the *ShortestPath* 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." — Carl Sagan

We can get a faster algorithm 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 SingleSourceDistances

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 $F = \{v \in V : v \notin S, \exists u \in F \text{ s.t. } (u, v) \in E\}$;
  For every $v \in F$, $\texttt{dist}_s[v] = d$;
  $S = S \cup F$;
return $dist_s$

Algorithm 1: BFS for SingleSourceDistances

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, and F the *frontier*, which is the set of vertices that were visited for the first time in the previous iteration. At each iteration, we update F by taking all vertices not previously visited that can be reached from the previous frontier by one additional edge. Then we add all of these new frontier vertices into S.

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?

Q: How can we implement BFS faster?

Putting all the above together, we obtain:

Theorem 4.1. $\texttt{BFS}(G)$ *correctly solves SingleSourceDistances 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?

Input	**:** A digragh $G = (V, E)$ and a source vertex $s \in V$
Queries	**:** For any query vertex t, return a shortest path from s to t (if one exists).

Data-Structure Problem SingleSourceShortestPaths

Theorem 5.1. *There is a solution that solves the SingleSourceShortestPaths problem on digraphs with n vertices and m edges with Preprocessing time $O(n+m)$ and the time to answer a query t is $O(\mathrm{dist}_G(s,t))$.*

It is natural and very useful (e.g. in Google maps) to have data structures for Shortest Paths 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 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(deg_out(v));
    v = j'th out-neighbor of v;
return ∞
```

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, ℓ) will visit all vertices reachable from s. Thus, we obtain a *Monte Carlo* algorithm for UndirectedSTConnectivity.

Theorem 6.1. *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.*