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; be sure to prepare and come on time.
- PS2 grades out; regrade requests due Fri night, revision videos due Sun night.
- Add-drop deadline Monday.
- Midterm 10/17 (no pset due 10/16)
- PS 2 feedback:
  - Median time 9hrs, 75th percentile 12hrs.
  - Good to have a summary when we finish a lecture/topic.
  - Manage time per student in OH.
- Salil's OH after class today; Salil away next week.
- Anurag OH on Zoom tomorrow (Fri) 1:30-2:30pm.

2 Graph Algorithms

Recommended Reading:

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

Motivating Problem: Another amazing success of algorithms we use every day are the map applications we use on our phones and web browsers, which find us the quickest route between any two destinations in the world in a fraction of a second. How is this possible? We will study the most basic formulations of this route-finding problem and get a taste of one of the main techniques that is used to solve it. Let us abstract the problem as follows:

> *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: Graphs!

Definition 2.1. A *directed graph* $G = (V, E)$ consists of a finite set of *vertices* V (sometime called *nodes*), and a set E of *edges*, which are ordered pairs (u, v) where $u, v \in V$ and $u \neq v$.

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

- Sometimes we allow *multigraphs*, where there can be more than one edge from u to v, and possibly also loops (edges $\{u, u\}$). Our definition as above is that of a *simple* digraph.
- We have defined an *unweighted* graph, but we may also want to assign weights/costs/lengths to each edge (e.g. modelling travel time on a road).
- An *undirected* graph has unordered edges $\{u, v\}$. (This models, for example, a road network where there are no one-way roads.) Equivalently, we can think of this as a directed graph where if $(u, v) \in E$, we also have $(v, u) \in E$.
- A graph is *planar* if it can be drawn in a 2D plane without edge crossings. Road networks are mostly but not entirely planar (e.g. consider overpasses).
- Some real-world graphs have some additional (e.g. hierarchical) structure that might be useful to exploit in algorithms (e.g. we may know that usually the best way to drive from one city to another is to use local roads to get to/from a highway).

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.

Representing Graphs. In this section of the course, we'll be studying many computational problems where the input is a graph G. We need to specify how the graph is encoded as an input to our algorithm. We will work with the *adjacency list* representation, where for each vertex v, we are given an out-neighbor array $\texttt{Nbr}_{\texttt{out}}[v]$ storing the elements of the set

$$\mathrm{N}_{out}(v) = \{u : (v, u) \in E\}.$$

The array $\texttt{Nbr}_{\texttt{out}}[v]$ is given together with its length, which is equal to

$$\deg_{out}(v) = |\mathrm{N}_{out}[v]|.$$

Thus, for $i \in [\deg_{out}(v)]$, $\texttt{Nbr}_{\texttt{out}}[v][i] = u_i$ where $u_0, \ldots, u_{\deg_{out}(v)-1}$ is an arbitrary ordering of the elements of $\mathrm{N}_{out}(v)$.

Analogously, the *in-neighbors* are $\mathrm{N}_{in}(v) = |\{u : (u, v) \in E\}|$ and the *in-degree* is $\deg_{in}(v) = |\mathrm{N}_{in}(v)|$, but we won't have those given explicitly as input to our algorithms. In undirected graphs, we will just write $N(v) = \mathrm{N}_{in}(v) = \mathrm{N}_{out}(v)$ for the *neighbors* of v and $\deg(v) = \deg_{in}(v) = \deg_{out}(v)$ for the *degree* of v.

Other ways of representing a graph, like the *adjacency matrix* are sometimes useful, and discussed in classes like AM 107 and CS 1240. In CS 1200, we'll always represent graphs by adjacency lists.

As throughout the course, we are working in the Word-RAM model, and we'll be using the standard word length of $w = \Theta(\log n)$ where n is the number of vertices in the graph. Recall that

word length at least $\log n$ is needed to avoid crashing when the input is loaded in (since a graph), and word length $w = O(\log n)$ always suffices to avoid crashing for polynomial-time algorithms (and this word length will also suffice for the exponential-time algorithms we will see, since they will only require $n^{O(1)}$ memory). Note that this means that the name of a vertex v can be stored 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 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 $i = 1, \ldots, \ell$.
- A walk in which all vertices are distinct is also called a *path*.
- The *length* of a walk w is length(w) = the number of edges in w (the number ℓ above).
- The *distance* from s to t in G is

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

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

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

A: As usual, Exhaustive Search! We 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 (i.e. w is a* path*).*

Proof. Suppose for contradiction that there is a shortest walk $w = (s = v_0, v_1, \ldots, v_\ell = t)$ 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 walk $w' = (s = v_0, \ldots, v_{i-1}, v_i = v_j, v_{j+1}, \ldots, v_\ell)$. We have the length of w' is strictly less than that of w and has the same start and endpoints. But then w is not a shortest walk, so we have a contradiction. □

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: $(n-1)! \cdot O(n) = O(n!)$

There is one choice for the first vertex, $n-1$ choices for the second vertex, $n-2$ choices for the third vertex, and so on, for a total of $(n-1)!$ possible paths. For each path, it takes $O(n)$ time to check that it is a correct path.

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.

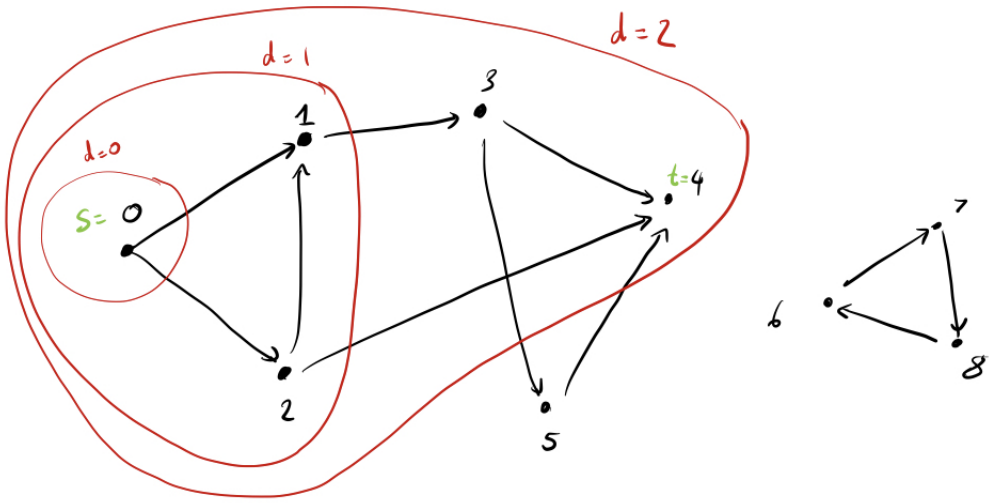
```
1 BFS(G, s)
  Input      : A digraph G = (V, E) and a source vertex s ∈ V
  Output     : The array dist_s[·] = dist_G(s, ·)
2 Initialize dist_s[t] = ∞ for all t ∈ V.;
3 S = F = {s};
4 dist_s[s] = 0;
5 foreach d = 1, ..., n − 1 do
6 |   Let F = {v ∈ V : v ∉ S, ∃u ∈ F s.t. (u, v) ∈ E};
7 |   For every v ∈ F, dist_s[v] = d;
8 |   S = S ∪ F;
9 return dist_s
```

Algorithm 1: BFS for SingleSourceDistances

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

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

For that graph, the lists of neighbors of the vertices are = [[1,2], [3], [1, 4], [4,5], [], [4], [7], [8], [6]].



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? For $d = 0, \ldots, n-1$, let S_d and F_d be the sets S and F at the start of iteration $d+1$ or, equivalently, the end of iteration d.

By induction on $d = 0, 1, \ldots, n-1$, we prove:

$$F_d = \{v \in V : \text{dist}_G(s, v) = d\}.$$

and

$$S_d = \{v \in V : \text{dist}_G(s, v) \leq d\},$$

Once proven, this implies that in iteration d we correctly set $\texttt{dist}_s[v] = d$ for every vertex v at distance d. Since all vertices either have distance at most $n-1$ or ∞ from s, the output array $\texttt{dist}_s$ is correct.

Base Case ($d = 0$): $S_0 = F_0 = \{s\}$, which is indeed the set of vertices at distance $d = 0$ from s.

Inductive Step: For $d \geq 1$, we have In the algorithm, we have:

$$\begin{aligned} F_d &= \{v \in V : v \notin S_{d-1}, \exists u \in F_{d-1} \text{ s.t. } (u, v) \in E\} \\ &= \{v \in V : \text{dist}_G(s, v) > d-1, \exists u \text{ s.t. } \text{dist}_G(s, u) = d-1, (u, v) \in E\} \qquad \text{(inductive hypothesis)} \\ &= \{v \in V : \text{dist}_G(s, v) = d\} \end{aligned}$$

The last equality is true because a vertex v is at distance d from s iff it can be reached by one edge from a vertex u at distance $d-1$ and there was no walk of length shorter than d to v. For S_d, we

have:

$$\begin{aligned} S_d &= S_{d-1} \cup F_d \\ &= \{v \in V : \text{dist}_G(s,v) \leq d-1\} \cup \{v \in V : \text{dist}_G(s,v) = d\} \qquad \text{(by inductive hypothesis and above)} \\ &= \{v \in V : \text{dist}_G(s,v) \leq d\}, \end{aligned}$$

as desired.

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

We have n iterations, and in each iteration, we can update F by enumerating over all edges $(u,v) \in E$, checking for each one whether $v \notin S$ and $u \in F$. In order to do these checks in constant time, we can store S and F as bitvectors, i.e. an arrays of n bits, where the u'th entry is 1 iff u is in the set. Then updating F takes $O(m)$ time and updating S as $S \cup F$ takes $O(n)$ time. This gives a total runtime $O(n(m+n))$.

Q: How can we implement BFS faster? Instead of enumerating over all edges $(u,v) \in E$ to update F, we only need to enumerate over ones where $u \in F$. Since each vertex u appears in F in at most one iteration, each edge (u,v) is only checked once across all iterations.

To achieve the benefit of this idea, we should store F not as a bitvector but as a linked list of vertices. Then we can enumerate over the vertices $u \in F$ in time $O(|F|)$, and for each we can enumerate over its adjacency list $\texttt{Nbr}_{\texttt{out}}[u]$ to see all vertices v that we should consider putting into F (after checking that $v \notin F$). Thus a single iteration starting with frontier F takes time:

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

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

$$\begin{aligned} O\left(\sum_{d=1}^{n-1}\sum_{u \in F_d}(1 + \deg_{out}(u))\right) &\leq O\left(\sum_{u \in V}(1 + \deg_{out}(u))\right) \\ &= O(n+m). \end{aligned}$$

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: We have presented BFS as doing "batch" updates to the frontier F, i.e. we go from the vertices at distanec $d-1$ to the vertices at distance d all together in Line 6. In practice F is maintained as a queue and vertices u are processed one at time. Initially the queue starts with vertex s. As long as the queue is nonempty, we remove the vertex u at the head of the queue, go over the vertices v in its adjacency list $\texttt{Nbr}_{\texttt{out}}[u]$, and for each one where $v \notin S$, we add v to both S and the end of the queue and set $\texttt{dist}_s[v] = \texttt{dist}_s[u] + 1$. This algorithm is equivalent to Algorithm 1, with a particular order of processing vetices.

A second implementation comment is that, since we are maintaining $\texttt{dist}_s$, it is redundant to also maintain S separately, since the algorithm maintains the invariant that a vertex v is in S iff $\texttt{dist}_s[v] < \infty$.

5 Finding the Paths

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

We can augment BFS to maintain an auxiliary array A_{pred} of size $|V|$, where A_{pred} holds the vertex u that we "discovered" v from. That is, if we add v to F because of the edge (u, v) (which would happen while going over the adjacency list of u), we set $A_{pred}[v] = u$. After the completion of BFS, we can reconstruct the path from s to t using this predecessor array, namely as $t, A_{pred}[t], A_{pred}[A_{pred}[t]], \ldots$ until we reach s.

This predeceessor array allows us to reconstruct the shortest path from s to t not only for a single vertex t but for any vertex t we desired. Thus, it solves the following (static) data structure problem:

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).

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 $O(\text{dist}_G(s, t))$.*

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

Corollary 5.2. *ShortestWalks (equivalently, ShortestPaths) can be solved in time $O(n + m)$.*

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: 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 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.*