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

Lecture 10: Graph Search

Harvard SEAS - Fall 2022 — *2022-10-06*

1 Announcements

Recommended Reading:

- Roughgarden II Sec 8.1–8.2
- CLRS 22.2
- Reminder: solutions posted on Canvas
- Pset 4 released
- Sender-Receiver exercise on Tuesday.
- Midterm feedback at https://tinyurl.com/cs120midtermfeedback

2 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 2.1. 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 2.2. *If w is a shortest walk from s to t, then all of the vertices that occur on w are distinct. That is, every shortest walk is a* path *— a walk in which all vertices are distinct.*

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. $\square$

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.

3 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:

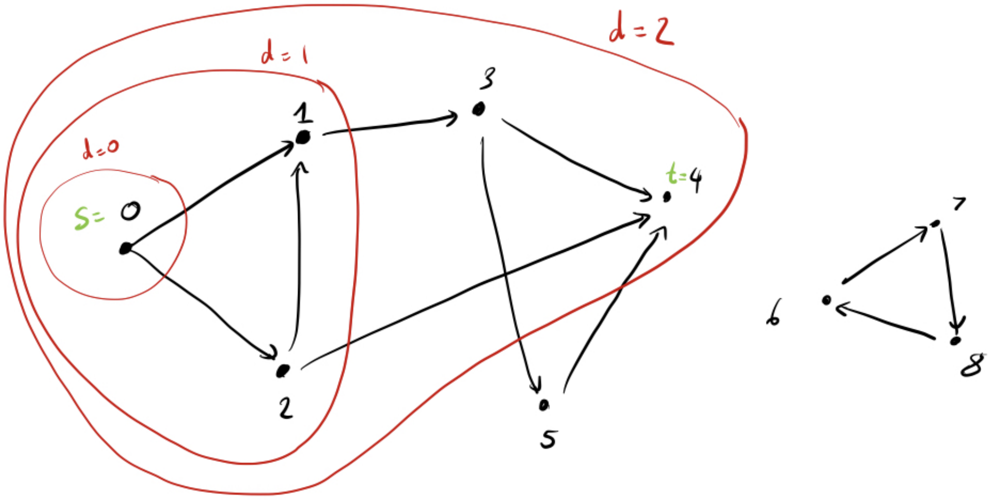
Input	: A directed graph $G = (V, E)$ and two vertices $s, t \in V$
Output	: The distance from s to t in G

Computational Problem DistanceInGraph

`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
$S = \{s\}$;
`/* loop invariant:` S `contains the vertices at distance` $\leq d$ `from` s `*/`
foreach $d = 0, \ldots, n-1$ **do**
  if $t \in S$ **then return** d;
  $S = S \cup \{v \in V : \exists u \in S \text{ s.t. } (u, v) \in E\}$
return ∞

Example:
Consider the graph with vertices $V = [9]$ and edges $E = \{(0,1), (0,2), (1,3), (2,1), (2,4), (3,4), (3,5), (4,5), (6,7), (7$

For that graph, the lists of neighbors of the vertices are `Nbr`= [[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. 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 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.

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 $d = 0$, $S = \{s\}$.

Inductive 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, in terms of the number of vertices n and the number of edges m?

[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.

We have n iterations, and in each iteration, we enumerate over all possible edges (u, v) in E. We can iterate over all possible edges by iterating over all n of the neighbor arrays, the sum of whose lengths is m, which takes time $O(m + n)$.

(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 $O(n(m + n))$.

4 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 $s, t \in V$
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` $\leq d$ `from` s`,` F `= vertices at distance` d `from` s `*/`
6 **while** $F \neq \emptyset$ **do**
7   **if** $t \in F$ **then return** d;
8     $F = \{v \in V - S : \exists u \in F \text{ s.t. } (u, v) \in E\}$;
9     $S = S \cup F$;
10     $d = d + 1$;
11 **return** ∞

Theorem 4.1. `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

$$
\begin{aligned}
O\left(\sum_{d=0}^{\infty} \sum_{u \in F_d} (1 + d_{out}(u))\right) &\le O\left(\sum_{u \in V} (1 + d_{out}(u))\right) \\
&= O\left(\sum_{u \in V} 1 + \sum_{v \in V : (u,v) \in E} 1\right) \\
&= O\left(n + \sum_{u \in V} \sum_{v \in V : (u,v) \in E(u)} 1\right) \\
&= O(n + m).
\end{aligned}
$$

□

Above we used the following definition:

Definition 4.2. 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?
The out-degree $d_{out}(v)$ is just the length of $\texttt{Nbr}(v)$. We specified that we stored the length of each such list as part of the representation of G, so we can just read that number.

5 More Graph Search

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

Note that, by Lemma 2.2, shortest walks are paths, so we can use the terms "shortest paths" and "shortest walks" interchangeably.

Maintain an auxiliary array A_{pred} of size $|V|$, where $A_{pred}[v]$ holds the vertex u that we "discovered" u from. That is, if we add v to the frontier when exploring the neighbors of u, set $A_{pred}[v] = u$. After the completion of BFS, we can reconstruct the path from s to t using this predecessor array.
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 5.1. *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.

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 $s, t \in V$, and a walk-length ℓ
Output : YES or NO
$v = s$;
foreach $i = 1, \ldots, \ell$ **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?
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.*