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).
- 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?

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 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* graph.
- 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 $\{u, v\}$. Equivalently, we can think of this as a directed graph where if $(u, v) \in E$, we also have $(v, u) \in E$. (We could think of this as a road network with no one-way roads.)
- 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.

Remark: as we'll see, 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.

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 $i = 1, \ldots, \ell$. A walk in which all vertices are distinct is also called a *path*.
- The *length* of a walk w is $\text{length}(w)$ = 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 $\text{length}(w) = \text{dist}_G(s,t)$

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.
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. □

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

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 digraph $G = (V, E)$ and two vertices $s, t \in V$
Output	**:** The distance from s to t in G

Computational Problem DistanceInGraph

How is the graph given to us? 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]

[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.

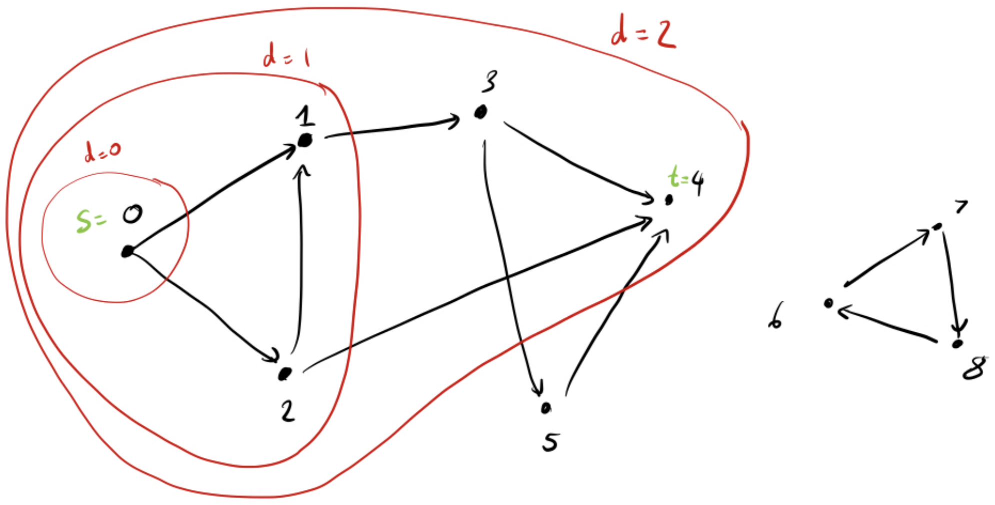
With this, here is the first version of BFS.

```
1 BFSv0($G, s, t$)
  Input  : A digraph $G = (V, E)$ and two vertices $s, t \in V$
  Output : The distance from $s$ to $t$ in $G$
2 $S = \{s\}$;
3 /* loop invariant:  $S$ contains the vertices at distance $\leq d$ from $s$  */
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 $\infty$
```

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 `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 update S by taking the union of current S with *the set of vertices that can be visited from all the vertices in S by one additional edge.*

Q: How is S stored?
In order to be able to check whether $u \in S$ and 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 $u \in S$.

Q: How do we perform the update of Line 6?

We'll iterate over all edges of G using the adjacency list and update the bitvector representation of S.

Q: How do we prove correctness?
Establish the loop invariant from start of iteration d.

$$S = \{v \in V : dist_G(s, v) \leq d\}$$

We now can prove that the loop invariant holds by induction on d.

Base Case: At $d = 0$, $S = \{s\}$.

Inductive Step: Let us denote by T the set maintained at iteration $d - 1$, for notational convenience. That is, $T = \{u \in V : dist_G(s, u) \leq d - 1\}$. Let S be the set at iteration d. We first argue that $S \subseteq \{v \in V : dist_G(s, v) \leq d\}$. Indeed, any v added in d th iteration has a $u \in T$ with $(u, v) \in E$. Since $dist_G(s, u) \leq d - 1$, we have $dist_G(s, v) \leq d$. On the other hand, we also have $\{v \in V : dist_G(s, v) \leq d\} \subseteq S$. For any vertex v with $dist_G(s, v) = d$, we have a vertex u such that $(u, v) \in E$ and $dist_G(s, u) = d - 1$. Thus, $u \in T$ and hence $v \in S$.

Then the loop invariant establishes that if the shortest path from s to t is of length k, we will add t 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?

We have n iterations, and in each iteration, we enumerate over all possible edges (u, v) in E - updating S accordingly. We can enumerate over all possible edges by going over all n 'neighbor' arrays, the sum of whose lengths is m. This takes time $O(m+n)$, which includes the time to update S.

This gives a total runtime $O(n(m + n))$.

5 Improving BFS

It will be useful in this section to take into account the number of adjacent edges to a vertex. Thus, we introduce the following definition.

Definition 5.1. 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 `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.

The improved BFS algorithm is based on the following 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 digraph $G = (V, E)$ 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      $F = \{v \in V - S : \exists u \in F$ s.t. $(u, v) \in E\}$;
9      $S = S \cup F$;
10     $d = d + 1$;
11 return $\infty$
```

Theorem 5.2. 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:
   We maintain F as a linked list (a queue of vertices). To carry out the update in Line 8, we enumerate over every *vertex* u in the frontier F, and try every edge (u, v) leaving that vertex (using the 'neighbour' array in the adjacency list). We check whether v lies in S - if v does not lie in S then update F accordingly [2]. 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 note here 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

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

[2] We do this by pushing v into the queue and eventually popping u out - we will ignore these details.

□

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.

Maintain an auxiliary array A_{pred} of size $|V|$, where $A_{pred}[v]$ holds the vertex u that we "discovered" v 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 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

```
1 RandomWalk($G, s, \ell$)
  Input  : A digraph $G = (V, E)$, a vertices $s, t \in V$, and a walk-length $\ell$
  Output : YES or NO
2 $v = s$;
3 foreach $i = 1, \ldots, \ell$ do
4     if $v = t$ then return $YES$;
5     $j = \texttt{random}(d_{out}(v))$;
6     $v = j$'th out-neighbor of $v$;
7 return $\infty$
```

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 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.*