CS 374: Additional notes on Graphs

Chandra Chekuri[*]
University of Illinois at Urbana-Champaign

March 18, 2017

1 Introduction

These notes are intended to supplement the lecture slides and existing notes and references on graphs available at our institution (in particular Jeff Erickson's excellent notes) and else where. I assume that the reader is already familiar with the basics of graphs and their representation via adjacency lists and adjaceny matrices. We will asssume throughout these notes that graphs are represented via adjacency lists. We will also assume that graphs are simple (no loops and parallel edges) unless we explicitly mention otherwise. For the most part n will denote the number of nodes/vertices and m will denote the number of edges of a graph.

Graphs have numerous applications in computer science and other areas inlcuding mathematics. Their power comes from their ability to model a wide variety of problems. This leads to a number of interesting algorithmic problems on graphs. There are two consequences of this phenomenon. First, an efficient algorithm for a basic graph problem leads to many applications. Second, some algorithmic problems on graphs turn out to b be intractable. From an algorithm designer's point of view the following skills are relevant when working with graphs.

- Ability to model a given problem as a problem on graphs.
- Knowledge of basic graph problems that admit efficient algorithms.
- Ability to understand simple variations of existing graph problems and be able to solve them either via reductions or simple changes to existing algorithms.
- Understanding structural properties of graphs that enable algorithmic developments and understanding.
- Knowledge of some intractable graph problems and ability to use simple reductions to establish intractability of other problems.
- Ability to develop new algorithms for graph problems.

Undirected vs Directed graphs: In most problems it is important to know whether the graph in question is an undirected graph or a directed graph. Typically (not always) directed graph problems tend to be more general, and hence also more difficult, than their undirected graph counterpart. For an undirected graph $G = (V, E)$ we define a *bi-directed* graph $\bar{G} = (V, E')$ as the graph obtained by replacing each undirected edge $uv \in E$ by two directed edges (u, v) and (v, u).

[*]chekuri@illinois.edu. Comments and corrections are welcome.

Representation and implementation issues: Graph algorithms and graph properties go hand in hand. Mathematically a graph $G = (V, E)$ is represented by a finite set of nodes V and a finite set of edges E. We use terminology such as u, v, w for nodes and e for edges in describing algorithms at a high-level or in proofs. One needs to be more careful when working with concrete implementation of algorithms and to figure out the actual running time. It is therefore useful to discuss a concrete representation. In the concrete representation $V = \{1, 2, \ldots, n\}$, that is, vertices are numbered 1 to n. We number edges from 1 to m. The graph is stored in two arrays. There is an array E of size m where $E[j]$ stores information about edge j including its two end points (which are integers from 1 to n) and other information such as weight or length depending on the application. We also have an array $Adj[1..n]$ where $Adj[i]$ has a pointer to the adjacency list of vertex i. In the case of undirected graphs we can assume that $Adj[i]$ is the list of all neighbors of i. In the case of directed graphs we will assume it is the list of all nodes to which i has edges to. It is often convenient to assume that $Adj[i]$ has a list of edges in the form of their numbers. Using the array E and this list we can easily obtain the list of neighbors. See Fig 1.

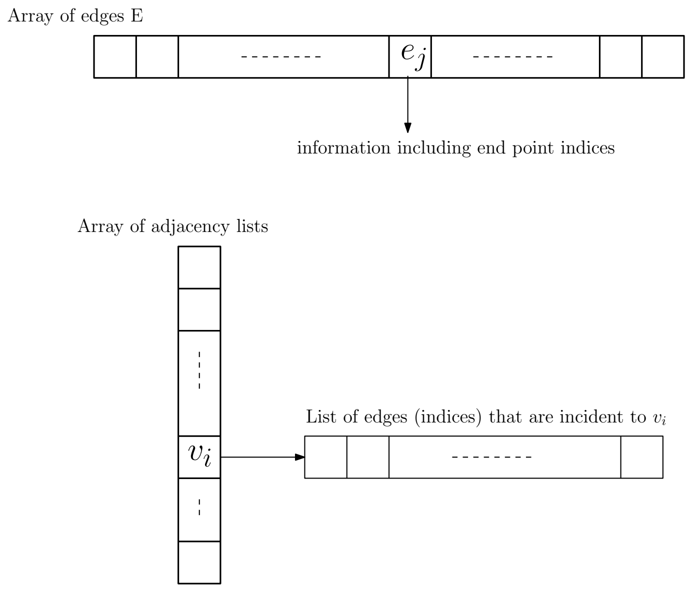
Array of edges E
e_j
information including end point indices
Array of adjacency lists
v_i
List of edges (indices) that are incident to v_i

Figure 1. Representing graphs via an array of edges and array of adjacency lists.

We note that the representation using arrays is convenient for graphs that do not undergo changes (static graphs). Pointer based list structures are more flexibe for for dynamic graph algorithms. We omit further discussion of these more advanced topics.

The representation size of a graph is $O(m + n)$ words. Note that we are working in the RAM model where we assume the word size in bits is sufficiently large to hold the number n^2 (so that edge indices also fit in a word). Technically we are assuming that the word size is $\Omega(\log n)$. When we say that an algorithm runs in linear time for a graph problem we mean that it runs in $O(n + m)$ time in the RAM model. When we discuss shortest path problems edges have lengths which we assume without loss of generality are integers. We will assume that each edgelength fits in one word, and that the computations during the algorithm do not result in a number that is much larger than the original numbers. Technically we are assuming that the word size is $\Omega(\log B + \log n)$ where B is the largest number in the input and that all numbers generated by the algorithm fit in $O(\log B + \log n)$ bits.

Definition 1.1. *For a directed graph $G = (V, E)$ the graph $G^{rev} = (V, E')$ is the* reverse *of G which* *is obtained by reversing the orientation of each edge of G.*

Question 1. *Given a representation of a directed graph $G = (V, E)$ describe how to obtain a representation of G^{rev} in $O(m + n)$ time.*

2 Reachability and Connectivity in Graphs

The most basic problem in graphs is the reachability problem.

Problem 1. *Given a graph $G = (V, E)$ and two distinct nodes s, t is there a a path from s to t?*

Before we proceed further we observe that in several settings walks are easier to deal with than paths. Recall that a *walk* allows vertices and edges to be repeated while a path does not. A *trail* is a walk where only vertices are allowed to be repeated and not edges. Note that if W is an s-t walk (or a trail) then there is an s-t path P all of whose vertices and edges are contained in W.

The preceding problem can be solved in linear time using basic graph traversal techniques including DFS and BFS. We note that the specific traversal method is not important to solve the preceding problem unless you desire more information. In fact the following problem can also be solved in linear time.

Problem 2. *Given a graph $G = (V, E)$ and a node s, find all nodes that have a path* from s.

Question 2. *Describe a linear time algorithm that given a directed graph $G = (V, E)$ and a node s, find all nodes that have a path* to s.

Remark 2.1. *The basic connectivity problems can be solved in undirected graphs and directed graphs by similar graph traversal algorithms but it is important to remember that there are important differences in the structural features and properties that they output.*

It is helpful to recall a basic property of connectivity. Let us define a relation C on $V \times V$ where uCv means that there is a path *from* u to v. We note that C is trasitive in both undirected and directed graphs: for $u, v, w \in V$, uCv and vCw implies uCw.

2.1 Connectivity properties in undirected graphs

In an undirected graph the connectivity releation C is symmetric (u has a path to v iff v has a path to u) and reflexive (uCu for each u) and transitive. This implies that C partitions V into *connected components*.

Problem 3. *Given an undirected graph $G = (V, E)$ compute all the connected components of G.*

Basic graph traversal algorithms solve the preceding problem in linear time. We also note that the traversal algorithms output a compact certificate of the components in the form of a forest with one spanning tree per component. Using these trees one can answer all the connectivity queries efficiently.

Question 3. *How should one store the connected component information in $O(n)$ space such that given any two node indices i and j one can answer in $O(1)$ time whether i and j are in the same connected component?*

Question 4. *Suppose you know m, n, and k the number of connected components of a graph but nothing else. Can you tell whether G has a cycle from this information? Can you tell whether G has at least two different cycles?*

Question 5. *Given an undirected graph $G = (V, E)$ describe a linear time algorithm that outputs a cycle in G if there is one or correctly reports that there is no cycle in G.*

Question 6. *Given an undirected graph $G = (V, E)$ and a node s describe a linear time algorithm that outputs a cycle in G that contains s if there is one, or correctly reports that there is no such cycle.*

2.2 Connectivity properties in directed graphs

Note that connectivity in directed graphs is asymmetric. uCv does not imply that vCu. Standard traversal algorithms allow us to find all the nodes reachable from a given node u in linear time. They produce a certificate in the form of a directed out-tree rooted at u. Using the reverse of the graph one can also compute the set of nodes that can reach a given node u.

When thinking about directed graphs a useful notion is that of *strong connectivity* which is a symmetric notion and is defined as follows: u is strongly-connected to v iff u can reach v and v can reach u. Strong connectivity induces an equivalence relation in directed graphs. Strong connected components are the equivalence classes of this relation. Note that each component needs to be *maximal*.

As we saw in lecture, given a directed graph G we can associate a new graph G^{SCC} called the meta-graph or the strong-connected component graph where the nodes of G^{SCC} are the strong connected components of G and we add edges between two components S_i and S_j if there is some $u \in S_i$ and $v \in S_j$ such that (u, v) is an edge in G. We observed that G^{SCC} is a directed acyclic graph (DAG for short). Note that if G is a DAG then G^{SCC} is G itself since each node of a DAG is a strong connected component. Thus DAGs cannot be simplified using this operation.

A simpe and useful property of DAGs is that they have a source (a node without incoming edges) and a sink (a node without outgoing edges).

DAGs have numerous applications because they model various situations in theory and practice. For instance, as we will see later, every recursive program on a given instances generates a DAG and this plays an important role in dynamic programming. Also, DAGs model partial orders.

A *topological sort* of a graph $G = (V, E)$ is an ordering or a permutation σ of the vertices $\{1, 2, \ldots, n\}$ (hence the ordering is $\sigma(1), \sigma(2), \ldots, \sigma(n)$) such that all the edges respect the ordering: that is, there is no edge $(\sigma(i), \sigma(j))$ in G where $i > j$.

We saw the following.

Lemma 2.2. *G admits a topological sort iff G is a DAG.*

Moreover we also discusssed two linear time algorithms for the following.

Problem 4. *Given a directed graph G decide if it is a DAG and if it is, output a topological sort.*

One is based on peeling off sources one at a time and the other is based on DFS. To make sure you understand the algorithms try the following question.

Question 7. *Describe a linar time algorithm that given a directed graph G outputs a cycle if it has one.*

Question 8. *Given an directed graph $G = (V, E)$ and a node s describe a linear time algorithm that outputs a cycle in G that contains s if there is one, or correctly reports that there is no such cycle.*

The following algorithmic problems can be solved in linear time by simply using the basic reachability algorithm. We discussed these in lecture.

- Given a directed graph G and a node u find the strongly connected component containing u.
- Given a directed graph G check if it is strongly connected.

A non-trivial problem is the following.

Problem 5. *Given a directed graph G compute all its strongly connected components and create G^{SCC}.*

We sketched in lecture a simple but very clever DFS based algorithm for the preceding problem.

2.3 DFS and BFS

DFS and BFS are two well-known graph traversal algorithms. Both are variants of a basic graph exploration strategy but have very different properties that can be expointed in algorithms. Exploiting DFS and BFS properties is more tricky than simply using black box graph traversal. For this reason it is useful to first think about the problem you need to solve and understand the desired properties, and then think about which graph traversal method may be most appropriate. For many applications using black box results is sufficient. For instance, you can simply use the fact that G^{SCC} can be obtained from G in linear time without specifically knowing or relying on the fact that it is based on DFS. Similarly, if you need shortest paths in an unweighted graph you can simply assume that BFS gives them to you rather than exploiting the actual details of BFS. Having said this, it is sometimes useful to know properties of the search tree that DFS and BFS generate.

DFS search generates pre and post-visit numbers that have some useful applications. DFS is helpful in exposing certain structural properties of graphs related to connectivity. In undirected graphs the search tree has the property that every non-tree edge connects two nodes in which one of the nodes is an ancestor of another. In other words there are no cross edges. This property is useful in finding cut edges and cut vertices of an undirected graph in linear time. In directed graphs the pricture is a bit more complex but, as we saw in lecture, we can exploit its properties to decide if a given graph is a DAG or not, and to find G^{SCC} from G.

Question 9. *Given an undirected graph G, suppose T is the search tree produced by $DFS(u)$. How can you tell if there is a cycle containing u? Same if G is directed.*

BFS helps find shortest paths in undirected graphs. BFS explores the vertices of the graph in increasing order of distance from the given start vertex s. We made this explicit by considering the layers version of the algorithm. Let L_i be the set of vertices at distance exactly i from s with $L_0 = \{s\}$. These layers have nice properties and so does the BFS tree. We saw that in undirected graphs there cannot be an edge whose end points u and v belong to layers L_i and L_j where $|i - j| > 1$. In directed graphs the situation is as follows. If (u, v) is an edge and $u \in L_i$ and $v \in L_j$ then $j - i \leq 1$. Note that j can be much smaller than i.

Question 10. *Let G be an undirected connected graph. Let T be a BFS tree rooted at s. Show that if there is an edge uv with both u and v in the same layer L_i (note that such an edge is not in T) then G has an odd-length cycle.*

Question 11. *Suppose G is an undirected graph and it contains an odd length cycle C. Then prove that in any BFS tree T there will be some edge uv of C such that u and v are in the same layer.*

One can use the preceding two claims to obtain a linear-time algorithm to check whether a graph G has an odd-length cycle. As a corollary, you can also and prove that G is a bipartite graph iff G has no odd length cycle. A graph is bipartite iff one can partition V into A and B such that all edges are between A and B. It is easy to see that if G is bipratite then it has no odd length cycle. The converse is harder to see but the preceding questions allow you to prove it.

Here is an interesting problem that is not so easy but gives you some idea of how undirected graphs and directed graphs can be different.

Question 12. *Suppose G is a simple connected undirected graph and let the maximum distance from a given node s be d. Prove that there is a node with degree at most $6n/d$. In particular if $d = \Omega(n)$ then there is a node of constant degree. Alternatively, if all degrees are large then the diameter of the graph cannot be too large.*

In contrast to the above the following can be shown.

Question 13. *Describe a directed graph G with diameter $\Omega(n)$ and such that the in-degree and the out-degree of each node is $\Omega(n)$. In other words the graph can be very dense and the diameter can still be almost as large as it can be.*

3 Graph Reductions: Modeling, Tricks, Proofs

This section is a small guide to solving problems on graphs using the basic algorithmic techniques of reachability, connectivity, topological sort etc. We illustrate a few tricks that are helpful. We list them here first and then give examples.

- Modeling problems via graphs. See examples of puzzles and games in Jeff's notes and exercises.
- Given a graph $G = (V, E)$ we can add a new node s and connect it to a subset $A \subset V$ of nodes. Reachability from s in the new graph allows us to compute reachability from A in G. Some time one needs to be careful with undirected graphs when using edge lengths. It is safer to use directed edges.
- Given a graph G we can work with G^{rev}.
- Use the meta-graph G^{SCC} to reduce a problem on a general directed graph to answering a similar or the same question on DAGs. DAGs are simpler and one can exploit the topological sort property.
- Given a graph we can create many copies of the vertices and choose appropriate edges between them to simulate various constraints. This idea of "layering" is related to the product construction for DFAs and is quite powerful. We will see some examples.

- DAGs model dependencies that have no cycles and are closely related to recursion and dynamic programming. This allows one to solve some problems via shortest path computations in DAGs bypassing the recursive approach. On the other side, several problems on DAGs can be solved via dynamic programming which can then be extended to general directed graphs via the reduction from G to G^{SCC}.

3.1 Modeling

Graphs can model several scenarios. A very high-level modeling approach is the following. Suppose we have a discrete system that evolves according some rules. We can model the *states/configurations* of the system via the nodes of a graph and add edges to model the *evolution* of the system. Reachability computation on this graph can be used to study whether one can reach a particular final state/configuration from some initial state/configuration. Many puzzles and games can be modeled this way including examples we saw in the lab and lectures.

We now consider some examples from material we saw earlier on DFAs. A DFA can be modeled as a graph where the states are the nodes and the transitions are the edges. Note that DFAs can have self-loops. Since we normally do not work with loops in our graphs, we need to handle them based on the problem at hand.

Reduce the following to a graph reachability problem.

Question 14. *Given a DFA $M = (Q, \Sigma, \delta, s, F)$ describe an algorithm to check if $L(M) = \emptyset$. What about checking whether $L(M) = \Sigma^*$?*

Question 15. *Given two DFAs $M_1 = (Q_1, \Sigma, \delta_1, s_1, F_1)$ and $M_1 = (Q_2, \Sigma, \delta_2, s_2, F_2)$ describe an algorithm to check if $L(M_1) \cap L(M_2) = \emptyset$.*

A more interesting question is the following.

Question 16. *Given a DFA $M = (Q, \Sigma, \delta, s, F)$ describe an algorithm to check if $L(M)$ is infinite.*

We give a hint. $L(M)$ is infinite iff the graph representing the DFA has a cycle (which could be simply a node with a self-loop) that is reachable from the start state and such that it reaches a final state. You need some care to handle the self-loops. A cleanly stated problem related to the preceding problem is the following.

Question 17. *Describe an efficient algorithm for the following problem: given directed graph $G = (V, E)$, two distinct nodes s, t and a subset $R \subseteq V$ of red nodes, is there a* walk *from s to t that contains a node from R?*

3.2 Reachability from a set of nodes

A common problem that arises in applications is the following simple generalization of the basic s-t reachability problem.

Problem 6. *Given a graph $G = (V, E)$ and two disjoint sets of nodes $A, B \subset V$, is there a path from some node in A to some node in B?*

To solve the preceding problem we add a new source node s and connect it to all the nodes in A with an edge (in the case of directed graphs the edge is directed from s to the nodes in A). We add a sink node t that has edges from each of the nodes in B. We then check whether s can reach t in the new graph G'.

A related problem is the following. How do you solve it?

Problem 7. *Given a graph $G = (V, E)$ and two disjoint sets of nodes $A, B \subset V$, is there a path for each node of B from some node in A?*

The preceding set-reachability problems arise indirectly or directly in other problems.

3.3 Reducing problem to DAGs

Recall that given a directed graph G one can compute G^{SCC} in linear time. Some interesting problems can be solved on a directed graph by first solving it on a DAG and then lifting the solution to a general graphs by reducing it to the DAG case. We saw some problems in the lab.

Here is another such problem.

Question 18. *Given a directed graph $G = (V, E)$ we say that a node u is* good *if u can reach every node in V. Describe a linear time algorithm to check if there is a good node in G.*

How do you solve the preceding problem in a DAG? Via a topological sort of course! Or by using properties of sources and sinks.

3.4 Product construction and layering

A powerful technique in graph algoritms is "product construction". We have already seen the notion of product construction in the context of DFAs where we were able to create a single DFA that simulates the behavior of two (or more) DFAs. A closely relate technique is that of creating multiple copies of a graph (or its vertex sets) which we call layers and connect them via some edges in an appropriate fashion to enforce some constraints.

We make this concrete in the context of some example problems. For the sake of brevity we give somewhat sketchy arguments which can be formalized with some additional effort.

Consider the following problem.

Problem 8. *Suppose $G = (V, E)$ is a directed graph where E is partitioned into red edges R and blue edges B. Is there an s-t path that where the edges in the path change color only once?*

Note that a desired path has a sequence of red edges followed by a sequence of blue edges or the other way around. This can be solved in linear time using first principles as follows. First let us check if there is an s-t path with red edges followed by blue edges. The other case is similar. We compute all nodes reachable from s via only red edges by removing all the blud edges. Let us call this set A. Similarly we can compute all the nodes B that can reach t using only blue edges. Call this set B. If $A \cap B$ is non-empty the answer is yes, otherwise no.

Here is an alternative algorithm. Create a new graph $G' = (V', E')$ as follows. $V' = V_1 \cup V_2$ where $V_1 = \{(v, 1) \mid v \in V\}$ and $V_2 = \{(v, 2) \mid v \in V\}$ are two copies of V. $E' = E_1 \cup E_2 \cup E_3$ where $E_1 = \{((u, 1), (v, 1)) \mid (u, v) \in R\}$ and $E_2 = \{((u, 2), (v, 2)) \mid (u, v) \in B\}$ and $E_3 = \{((u, 1), (u, 2)) \mid u \in V\}$. Thus (V_1, E_1) is is a copy of G with only red edges and (V_2, E_2) is a copy of G with only blue edges and the edges in E_3 connect the first copy of a vertex u to its second copy.

Claim 1. *There is path from $(s, 1)$ to $(t, 2)$ in G' iff there is an s-t path P in G where the edges in P change color at most once from red to blue.*

Thus, we have reduced the problem to a standard reachability problem in a new graph.

Next we consider another problem using a similar approach.

Problem 9. *Given a directed graph $G = (V, E)$ and two nodes s, t and an integer k, is there a walk from s to t with at most k edges?*

Of course one can solve the above problem by computing a shortest path from s to t using BFS and checking if its length is at most k. Here we will use another method which is not as efficient but illustrates a technique that will be useful in other contexts.

Given $G = (V, E)$ we create a new graph $G' = (V', E')$ with $k + 1$ layers of vertices where $V_i = \{(v, i) \mid v \in V\}$ is a copy of V in layer i. Formally, $V' = V \times \{0, 1, 2, \ldots, k\}$. We add the following edges: if (u, v) is an edge in G then we add for each $0 \le i < k$ the edge $((u, i), (v, i+1))$. In other words we connect the copy of u in layer i to the copy of v in layer $i + 1$ iff (u, v) is an edge of G. More formally $E' = \{((u, i), (v, i + 1)) \mid (u, v) \in E, 0 \le i < k\}$.

Claim 2. *There is an s-t walk in G with* exactly *k edges iff there is a path from $(s, 0)$ to (t, k) in G'. There is an s-t walk in G with* at most *k edges iff there is a path in G' from $(s, 0)$ to a vertex in the set $T = \{(t, i) \mid 1 \le i \le k\}$.*

The preceding claim reduced our original shortest path problem to a reachability problem in a simple way. We already remarked that the resulting algorithm is not as efficient as BFS, and hence it may appear that we have not gained much. However, we will see that the idea of layering has several powerful applications. Answer the following questions using the idea of layering.

Question 19. *Let $G = (V, E)$ be a directed graph whose edge set is partitioned into red edges R and blue edges B. Describe an efficient algorithm to check if there is an s-t walk in G in which the edges alternate in color.*

A related question is the following.

Question 20. *Let $G = (V, E)$ be a directed graph and some of the edges are red and some of the edges are blue ($R \subset E$ represents the red edges and $B \subset E$ represents blue edges). Describe an efficient algorithm to check if there is an s-t walk in G where in each prefix of the walk the difference between the number of red and blue edges is at most* 1.

3.5 Proving correctness of graph algorithms and reductions

As we have seen already a number of interesting problems on graphs can be reduced to solving one or more problems that have been previously solved such as rechability, computing connected components, topological sort etc. The basic technique that we rely on here is the notion of *reducing* one problem to another.

Formally a reduction from a problem $\mathscr{A}$ to a problem $\mathscr{B}$ is a mapping f that takes an instance I of $\mathscr{A}$ and produces an instance I' of $\mathscr{B}$. The idea is then to use an algorithm for $\mathscr{B}$ on I' and use the solution to obtain a solution for I. Thus, the algorithm for $\mathscr{A}$ consists of three steps:

- On input I to $\mathscr{A}$, use the mapping function f to generate I', an instance of $\mathscr{B}$.
- Run the algortim for $\mathscr{B}$ on I' to produced a solution S' for I'.
- Convert solution S' for I' into a solution S for I using a mapping f'.

A simpler setting is obtained if restrict attention to *decision* problems where the answer is yes or no. In this case the main step is the transformation f. Most reductions for decision problems fall into the simple category where the the answer to I is yes if the answer to I' is yes and the answer to I is no if the answer to I' is no. Thus, we only need to prove that the reduction f is correct. It boils down to showing that $f(I)$ has a yes answer iff I has a yes answer. We need to prove two directions.

We illustrate this with a simple example of a problem we have already seen, namely Problem 6. Given directed graph $G = (V, E)$ and two disjoint node subsets $A, B \subset V$ we need to check whether there is a path from some node in A to some node in B. To solve this we created a new graph $G' = (V \cup \{s, t\}, E')$ by adding two new nodes s, t and some edges as follows. $E' = E \cup \{(s, u) \mid u \in A\} \cup \{(v, t) \mid v \in B\}$. We then solve the s-t rechability problem in G' and return its answer. Note that we have a decision problem. To prove correctness of this reduction we need to show that there is a path from A to B in G iff there is a path from s to t in G'. Note that the iff means that we have to consider the two directions separately.

We do the if direction first. Suppose there is a path P from A to B. By this we meant that there is some $u \in A$ and some $v \in B$ and a path $P = u = a_1, a_2, \ldots, a_k = v$ that connects u to v in G. By construction of G' the edges (s, u) and (v, t) are in G' along with all the edges of G. Hence the path $P' = s, u = a_1, a_2, \ldots, a_k = v, t$ is path in G' that connects s to t in G'.

For the only if direction, suppose there is a path Q from s to t in G'. By construction of G', the first edge of Q must be (s, u) for some $u \in A$ since the only edges leaving s are to A. Similarly the last edge of Q is (v, t) for some $v \in B$. Thus Q is of the form $s, u, a_1, a_2, \ldots, a_k, v, t$ where $a_1, a_2, \ldots, a_k \in V$ since s has no incoming edges and t has no outgoing edges. Therefore, $u, a_1, a_2, \ldots, a_k, v$ is a path in G which implies that there is a path from A to B in G.

The preceding proof may appear to be "obvious" and in a sense it is. On the other hand there will be more complicated problems where checking *both* directions is *very important* to avoid incorrect reductions or algorithms. It is also worth mentioning that modifying existing algorithms such as DFS, BFS or Dijkstra in complicated ways often makes it hard to prove correctness. Even if one wants to do that, it may be worthwhile to understand what the modification is accomplishing, and rewrite the algorithm via a reduction to an existing problem.

4 Shortest Paths and Related Problems

In the shortest path problem we are given a graph $G = (V, E)$ and each edge $e \in E$ has a *length* $\ell(e)$. The goal is to find paths where the objective is to *minimize* the length of the path. There are two high-level problems of interest.

- Single-source shortest path problem: here we are given a node s and we want to find the shortest path length from s to all the nodes of G. In particular we may be interested in only one destination node t.
- All-pairs shortest path problem: we want to find the shrotest path length for each pair (u, v) of nodes.

There is a big distinction between the case when edge lengths are non-negative and when the edge-lengths can be negative. For the most part we will focus on edge-lengths being non-negative. In fact it turns out to be helpful to consider shortest walks instead of shortest paths. Note that when edge lengths are non-negative a shortest walk can be assumed to be a shortest path without loss of generality (why?). However, it is not true when edge lengths are negative (why?).

A key lemma on shortest paths is the following.

Lemma 4.1. *Let $G = (V, E)$ be a graph with edge lengths. Suppose $P = v_1, v_2, \ldots, v_k$ is a shortest path/walk from v_1 to v_k. Then for any $1 \leq i \leq j \leq k$ the path/walk $v_i, v_{i+1}, \ldots, v_j$ is a shortest path/walk from v_i to v_j.*

However, the following lemma is true only if the edge lengths are non-negative.

Lemma 4.2. *Let $G = (V, E)$ be a graph with non-negative edge lengths. Suppose $P = v_1, v_2, \ldots, v_k$ is a shortest path/walk from v_1 to v_k. Then for any $1 \leq i \leq j \leq k$ the* length *of the path/walk $v_i, v_{i+1}, \ldots, v_j$ is no more than the* length *of P.*

Corollary 4.3. *Let $G = (V, E)$ be a graph and let s be any node. Let $s = v_1, v_2, \ldots, v_n$ be a sorting of nodes of V in increasing order of distance from s (ties broken arbitrarily). Then for each $i > 1$ there is a shortest path from s to v_i such that all the intermediate nodes are from $\{v_2, \ldots, v_{i-1}\}$.*

Note that the preceding lemma is *false* if edges can have negative lengths!

4.1 Single-source Shortest Paths

First we consider the case of non-negative lengths. As we know BFS can be used to find shortest path lengths in both undirected and directed graphs in linear time where the assumption is that each edge has length 1. The key property that BFS exploits is Corollary 4.3. It inductively finds nodes in layers 0 to $i-1$ and then finds nodes in layer i by simply considering nodes adjacent to layer $i-1$ that are not in layers 0 to $i-1$. In other words it finds nodes of G in increasing order of distance from the start node.

Dijkstra's algorithm can be seen as an efficient implementation on BFS in a graph obtained by adding dummy nodes on each edge of length $\ell(e)$ to make all edge lenghts equal to 1. Alternatively, Dijkstra's algorithm is finding nodes in increasing order of distance from s by exploiting Corollary 4.3. Some books consider Dijkstra's algorithm a *greedy* algorithm. I strongly disagree. What is important in an algorithm is not its final description but how it was derived and how one proves its correctness. We do not think of BFS as a greedy algorithm but there is essentially no difference in the principles that help us understand BFS and Dijkstra's algorithm.

Important things to remember about single-source shortest path computation using BFS or Dijkstra's algorithm.

- Shortest path algorithms in undirected graphs when edges have non-negative lengths can be reduced to shortes path problem in directed graphs via the bi-directed graph. Nevertheless, it is useful to keep in mind that undirected graphs and directed graphs can behave differently.

- BFS returns a shortest path tree rooted at s in $O(n + m)$ time. Note that a shortest path tree is a compact representation of all shortest paths from s.

- Dijkstra's algorithm gives shortest path tree rooted at s in $O((n + m) \log n)$ time using standard priority queues and in $O(n \log n + m)$ time using Fibonacci heaps.

- Shortest paths *to* s can be obtained by reversing the graph.

- Triangle inequality: $d(u, v) + d(v, w) \geq d(u, w)$ for $u, v, w \in V$.

Here are few questions to help you think about shortest paths.

Question 21. *Suppose we want to find a shortest s-t path where the goal is to minimize the L_2 norm of the edge lengths on the path. That is, given a path P with edges $e_1, e_2, \ldots, e_k$ the L_2 norm is $\sqrt{\sum_{i=1}^{k} \ell(e_i)^2}$. What about L_p norm for some $p > 0$ defined as $(\sum_{i=1}^{k} \ell(e_i)^p)^{1/p}$? Note that the regular shortest path problem is to find the L_1 norm of the edge lengths.*

A question related to the above is the following. The L_∞ norm of the edge lengths of the path can be seen to be the maximum edge length in the path. Typically it is called the *bottleneck* length of a path. Formally the bottleneck length of a path P is $\max_{e \in P} \ell(e)$.

Question 22. *Adapt Dijkstra's algorithm to find the bottleneck shortest path distance from s to all the nodes of G.*

Question 23. *Given a directed graph G with non-negative edge lengths, using the layering technique find a shortest path from s to t among all paths with at most k edges.*

4.1.1 Negative edge-lengths

Negative edge-lengths are more complicated to deal with because the principles that drive BFS and Dijkstra's algorithm no longer hold. An important issue here is that if G has negative length cycle then the shortest walk length from s to t can be $-\infty$ because the walk can go around the cycle arbitrarily many times. One can of course for the shortest path intead of asking for a shortest walk. However, finding the shortest s-t simple path in a graph with negative edge lengths is equivalent to asking for the longest s-t simple path in a graph with non-negative edge lengths and the latter problem is NP-Hard (equivalent to the Hamiltonian Path/Cycle problems that we will see later). Thus we settle for the following.

Problem 10. *Given a directed graph $G = (V, E)$ with potentially negative lengths and nodes s, t, find the length of the shortest walk from s to t if it is finite, or report that it is $-\infty$ and find a negative length cycle in G.*

Implicit in the preceding problem is the following: the shortest s-t walk length in G is $-\infty$ only if there is a negative length cycle in G. Prove this fact.

Remark 4.4. *Note that there can be a negative length cycle in G that is not rechable from s or reach t. To detect a negative length cycle in G we can add new node s' to the graph and connect it via directed edges to all the nodes in V (and assign edge length 0 to each of these edges). Call this new graph G'. Computing shortest walk lengths from s' in G' helps detect negative length cycles in G.*

Remark 4.5. *Unlike the case of non-negative lengths, the problem of finding shortest walks with negative lengths in undirected graphs* cannot *be reduced to the corresponding problem in directed graphs via the bi-directed graph (why?). One needs more advanced techniques related to minimum cost perfect matchings which is beyond the scope of these notes.*

We saw in lecture that Problem 10 can be solved in directed graphs via the Bellman-Ford algorithm in $O(mn)$ time. There are two ways to understand Bellman-Ford. One is via dynamic programming. The other is via the layering technique. They are essentially the same but some prefer one to the other. The essential idea is that layering or DP can solve the following problem.

Lemma 4.6. *Given a directed graph G with potentially negative length edges there is an algorithm that runs in $O(mk)$ time and finds the shortest walk length from s to t for all $t \in V$ where the walks are constrained to have at most k edges.*

The space requirement is $O(m)$ even though the natural DP or the layering require $O(mk)$ space — one can see this as improving the space for DP.

The second observation which is useful is the following.

Lemma 4.7. *If there is a negative length cycle reachable from s in G then there is some node $v \in V$ such that $dist(s, v, n) < dist(s, v, n-1)$ where $dist(s, v, i)$ is the shortest walk length from s to v with at most i edges.*

The preceding two lemmas lead to the Bellman-Ford algorithm that we covered in lecture.

Question 24. *Describe how to modify the standard Bellman-Ford algorithm to compute a negative length cycle (and not just detect its existence) if there is one.*

4.1.2 Shortest paths in DAGs

As we saw DAGs are very relevant in understanding directed graphs. Since DAGs have no cycles shortest path distances are well-defined even when the graph has negative lengths. Thus, in DAGs, we can also compute longest simple paths. Also, using topological sort we can compute the shortest path distances in linear time. One can view this algorithm as a dynamic programming algorithm.

Lemma 4.8. *Given a DAG $G = (V, E)$ with arbitrary edge lengths and a node $s \in V$, one can find a shortest path tree rooted at s in $O(m + n)$ time.*

This algorithm has a number of applications in dynamic programming as we will see in a later section.

4.1.3 Local improvement algorithms

We have seen algorithms for shortest path problems that are built upon structural understanding of distance properties in graphs. There is another approach to finding shortest paths which is based on a local improvement step. Here we focus on directed graphs and also assume without loss of generality that s can reach all the nodes in V. To motivate the algorithms here, consider the following problem. Suppose we are told that a tree T rooted at s is a shortest path tree in G. How do we verify that this is true? Of course we can compute the shortest path distances from s using one of our favorite algorithms and then check if the distances are indeed the same as the ones given by T. Can we do better? Indeed there is a linear time algorithm and it is based on the following lemma.

Lemma 4.9. *Given an edge-weighted directed graph $G = (V, E)$ and a node s let T be a spanning directed out-tree rooted at s. Let $d_T(s, v)$ be the shortest path distance from s to v in T. T is a shortest path tree in G iff there is no edge $(u, v) \in E$ such that $d_T(s, v) > d_T(s, u) + \ell(u, v)$.*

It is clear that T is not a shortest path tree if there is an edge (u, v) such that $d_T(s, v) > d_T(s, u) + \ell(u, v)$. The lemma says that one can always find such an edge if T is not a shortest path tree. Assuming the lemma there is a simple $O(m)$ time algorithm to check whether T is a shortest path tree. Note that we have not assumed that edge lengths are non-negative.

We now prove the lemma which is not so obvious.

Proof: Suppose T is not a shortest path tree. This implies that there is some node v such that $d_T(s,v) > d_G(s,v)$ where $d_G(s,v)$ is the true shortest path distance. Among all such nodes v choose the one v^* whose shortest path from s has the fewest number of edges. Let k be the number of edges in this shortest path P from s to v^*. $P = s = u_1, u_2, u_2, \ldots, u_{k-1}, v$. Note that $P' = s, u_2, \ldots, u_{k-1}$ is a shortest path from s to u_{k-1}. By choice of v^* it must be the case that $d_T(s, u_{k-1}) = d_G(s, u_{k-1})$. We claim that (u_{k-1}, v^*) is an edge that witnesses the fact that T is not a shortest path tree. To see this we note that

$$d_T(s,v^*) > d_G(s,v^*) = d_G(s,u_{k-1}) + \ell(u_{k-1},v^*) = d_T(s,u_{k-1}) + \ell(u_{k-1},v^*).$$

□

The implication of the preceding lemma is that if we start with an arbitrary tree T as the shortest path tree then we can either realize that it is a valid shortest path tree or find an edge (u,v) to “improve” it. If T is not valid, we can change it by removing the edge into v in T and adding (u,v) to T. This decreases the distance to v and potentially other nodes. It not so obvious that repeated improvements would lead to a correct tree in a short amount of time. It turns out that with some care one can indeed achieve this. Jeff’s notes explore this as a generic strategy for shortest path problems. In his terminology, an edge is *tense* if it violates the current distances (according to T). He refers to the improvement operation as a *relax* operation. One can interpret Dijkstra’s algorithm and Bellman-Ford algorithm and many others in this local-improvement framework. However, I believe that understanding the principles that lead to the design of algorithms for shortest path algorithms for non-negative lengths and for negative lengths is important and it is actually helpful to think about them separately.

4.2 All-pairs Shortest Paths

In the all-pairs shortest path problem (APSP for short) we want to compute $d_G(u,v)$ for each pair of nodes (u,v). We can do this by running the single-source algorithm n times, one from each of the nodes of G. If edges have non-negative lengths then running Dijkstra’s algorithm n times give us an algorithm with running time $O(n^2 \log n + mn)$. Improving this run-time for APSP is a major open problem. If all edge lengths are 1 we can run BFS n times to obtain a run time of $O(mn)$. One can beat this run time for dense graphs by using fast matrix multiplication, however the resulting algorithm is not very practical. Nevertheless, it is a striking theoretical improvement.

When G has negative-length edges we already saw that Bellman-Ford can detect whether there is a negative length cycle in $O(mn)$ time. Suppose G has no negative length cycle. Then APSP is well-defined. However, running Bellman-Ford n times would result in a running time of $O(mn^2)$. There is a clever trick to reduce this to $O(mn)$. This is called Johnson’s algorithm and you can read Jeff’s notes for this. The trick is to do one Bellman-Ford from a single node s in $O(mn)$ time and then use the distances computed from s as *potentials* (since G has no negative length cycles). The potentials can be used to reduce the problem to a graph with non-negative edge lengths via the notion of reduced costs/lengths.

Finally, there is a very different algorithm for APSP called the Flloyd-Warshall algorithm which is based on a clever dynamic programming idea. This algorithm runs in $O(n^3)$ time irrespective of the number of edges. We discussed this is lecture. Although the running time is not better than $O(mn)$ the DP idea behind this algorithm is quite useful and finds other applications. In particular one can derive an algorithm to convert an NFA to a regular expression.

5 DAGs and Dynamic Programming

There is a close connection between DAGs and dynamic programming. We explain this connection here.

Recall that every DAG has a topological sort. Without loss of generality let $v_1, v_2, \ldots, v_n$ be a topological sort of a DAG $G = (V, E)$. If we want to find shortest paths from $s = v_1$ we have the following recurrence with base case $d(s, v_1) = 0$:

$$d(s, v_i) = \min_{j<i,(v_j,v_i)\in E} d(s, v_j) + \ell(v_j, v_i).$$

With appropriate memoization you see that all the distances from s can be computed in $O(n+m)$ time. We can also see Bellman-Ford algorithm as computing shortest paths in a DAG since the layered graph is in fact a DAG. Thus, DP can be used to solve shortest path problmes in DAGs and also general graphs.

DAGs in DP: What about the reverse connection? Why are DAGs useful in DP? To see this we consider a recursive program/algorithm $\mathscr{A}()$. Suppose we run $\mathscr{A}(I)$ on some instance I. What happens? The execution of $\mathscr{A}(I)$ generates a recursion *tree* T where each node in the tree corresponds to a subproblem I' and the leaves correspond to base cases. In divide-and-conquer type recursions the subproblems in the tree T are all different and there is no value in memoization. However, as we saw in dynamic programming, there are many interesting recursive programs/algorithms where the same sub-problem I' is generated many times in the tree T. This can lead to a tree T whose size is exponential in the size of the initial input I even though the number of *distinct* sub-problems is only polynomial in the size of I. In those case we can memoize the recursion to avoid recomputing the same problem again and again and we will obtain a polynomial-time algorithm. This is the essence of DP.

Let us examine $\mathscr{A}(I)$ and say $I_1 = I, I_2, \ldots, I_m$ are the *distinct* sub-problems that are generated in the tree T. We can crate a DAG $G(I)$ as follows. Create a node v_i for each I_i. Therefore the number of nodes in this graph is m. $\mathscr{A}(I_i)$ generates some subproblems, say, $I_{j_1}, I_{j_2}, \ldots, I_{j_h}$ and these will be its children in the recursion tree T. In other words I_i *depends* on calls to $I_{j_1}, \ldots, I_{j_h}$. We can encode this dependence by adding directed edges $(v_{j_1}, v_i), (v_{j_2}, v_i), \ldots, (v_{j_h}, v_i)$ to $G(I)$. Thus, we end up with a directed graph.

Claim 3. *$G(I)$ is a DAG.*

Why is the claim true? If $G(I)$ is not a DAG then there is circular dependency in the recursion and $\mathscr{A}(I)$ would not terminate! Now we know that $G(I)$ is a DAG we can do a topological sort of $G(I)$ and we observe that *any* topological sort gives us a way to order the evaluation of the sub-problems such that a subproblem I_i is evaluated only after all of the subproblems it depends on are already evaluated. Thus, if we draw the dependency graph for the sub-problems in a recursive algorithm we can *automate* the evaluation order by doing a topological sort. However, in DP we insist on explicitly specifying an evaluation order. This is mainly to make sure that we understand the structure of the specific problem at hand and can optimize for time and space using data structures.

Perhaps another advantage of the connection between DAGs and DP is that for some problems it is possible to reduce the problem directly to a shortest path computation in a DAG (or simply a reachability computation). This way of thinking is helpful and/or is natural for some. For instance the layering approach to solve Problems 9 and 10 does not explicitly refer to DP although it is implicitly present. We give an example. Recall that problem of checking whether a string w is in L^* given a sub-routine that checks if a given string x is in L. In other words we want to decide whether w can be split into strings in L. Below we describe a reduction to a path problem in DAGs.

Let $w = w_1 w_2 \ldots w_n$. Create a graph $G = (V, E)$ with n nodes $v_1, v_2, \ldots, v_n$. For $i \leq j$, add an edge (v_i, v_j) if the string $w_i w_{i+1} \ldots w_j$ is in L. Note that this is a DAG with $v_1, v_2, \ldots, v_n$ as the topological sort. We claim that $w \in L^*$ iff there is a path from v_1 to v_n in this DAG. We leave the proof of this as an exercise.

6 Summary of Problems/Algorithms

Here we list the basic graph problems and algorithms that we covered and are useful in solving related problems.

- Reachability in graphs. Given G and nodes s, t is there a path from s to t?
- Connectivity structure of graphs. In undirected graphs the connected components and a spanning forest structure. In directed graphs the meta-graph G^{SCC}.
- Topological sort for DAGs.
- Detecting cycles in graphs.
- Single-source shortest paths in graphs. BFS for unweighted, and Dijkstra for non-negative lengths. Bellman-Ford for negative lengths in directed graphs and negative cycle detection.
- Reduction techniques for graph problems.
- DAGs and dynamic programming.