Graph Algorithms (Updated)

These notes pick up from page 8 of the notes distributed at the previous class, more or less. This handout replaces that part of the previous handout.

- Graph Traversals Revisited
- Path Finding
  - Spanning Trees
  - Dijkstra’s Algorithm
- Kruskal’s Algorithm

Graph Traversals (Briefly) Revisited

Some aspects of the generic graph traversal algorithm presented in the last lecture caused unforeseen and unintended confusion. We suspect that the error was in presenting the algorithm in a form that was too generic and not suited for any particular purpose, and then trying to describe the more specific graph traversals in terms of this generic traversal.

So, we will restate the basic algorithms we use, in their entirety, in an attempt to clear up this potential source of confusion.
Connected Graph Traversal

Purpose: to visit every vertex connected to the start vertex exactly once.

1. Let \( s \) be some vertex. This is the start vertex - perhaps chosen arbitrarily.

2. Let \( S \) be the set of "active" vertices, initially containing only \( s \).

3. Loop until \( S \) is empty.
   a. SELECT and REMOVE a vertex \( v \) from \( S \).
   b. MARK \( v \) as visited.
   c. PROCESS \( v \) in whatever way appropriate.
   d. ADD all the unvisited vertices adjacent to \( v \) to \( S \).

   Note that since \( S \) is defined here to be a set, it cannot contain duplicate items. If \( S \) is actually implemented by a data structure that allows duplicates, then this must be taken into account. An easy way to do so is to mark all the vertices in \( S \), and never add a marked vertex to \( S \).

How \( v \) is SELECT’d (step 3.a), and/or how the vertices are ADD’d to \( S \) (step 3.d) has a profound effect on the order that vertices are visited. (a stack gives depth-first, a queue gives breadth-first)
Path Finding Algorithm

Purpose: to find a simple path from \( x \) to \( y \), or prove that no such path exists.

1. Let \( s \) be the zero-length path from \( x \) to itself.
2. \( S \) is a set of paths, initially containing only \( s \).
3. Loop. If \( S \) is Empty return \text{NIL}.
   a. Select a path \( p \) from \( S \), and Remove \( p \) from \( S \).
      Let \( v \) be the last vertex in \( p \). If \( v == y \), return \( p \).
   b. Mark \( v \) as visited.
   c. Process \( v \) or \( p \) (if our purpose is to process vertices or paths).
      d. Extend \( p \) by creating new paths (that begin with \( p \) but have one more edge added to the end) for each unvisited vertex adjacent to \( v \).
         Add each of these new paths to \( S \).

Note: depending on our application, we might actually add all of these paths, or we might only paths that end in a vertex that does not already appear anywhere in any path in \( S \).

If we are only looking for a path, then there is no reason for a vertex to appear more than once in any path in \( S \). For other purposes, we might need to consider multiple paths through the same vertex.
Minimum Path Finding

As mentioned earlier, using a queue in our path-finding algorithm gives us a *shortest* path - a path with the fewest possible edges.

If we are searching for a path in a network, however, the value of each edge may represent a *cost* associated with traversing that edge. In this case, we might desire to find the *minimum* path - the path that has the smallest total cost (although perhaps not the minimum number of edges).

- The algorithm we will study to find this is called **Dijkstra’s Algorithm**. Dijkstra’s algorithm is a natural extension of our path-finding algorithm.

- Instead of using a queue, however, we will use a **PRIORITY QUEUE** where the priority of a path is its total cost. We will always select the path with the *smallest* total cost from S.

- When adding paths to S, we will add *all* paths, not only paths to vertices not already in S. Even if we already have a path to v, a new path might be less expensive.

  Possible optimization - add only paths to vertices that are less costly than existing paths to those vertices.) Whether this actually helps depends on how the paths are selected.

*See Weiss for implementation notes.*
Dijkstra’s Algorithm - Intuitions

There are two basic ways to visualize Dijkstra’s algorithm. We will do both.

In the first method, the graph traversal aspect of the algorithm is hidden, and everything proceeds according to a table. The table has a row for each vertex, and columns for each of the following values:

- Whether the minimum cost path to that vertex from the start vertex is known (if we’ve proven a less expensive path can’t possibly exist).
- The cost of the minimum path from the start vertex to the vertex. Initially, this will contain a cost of infinity (so any valid path will cost less).
- The previous vertex on the minimum path from the start vertex to the vertex. (If there could be more than one such vertex, because there could be more than one such path - it doesn’t matter which we choose).
Dijkstra’s Algorithm - Example

\[ V = \{A, B, C, D, E\} \]
\[ E = \{(A, B, 4), (A, C, 1), (A, E, 2), (B, E, 5), (B, D, 1), (D, E, 8), (C, D, 12), (C, E, 3)\} \]

Our task is to find a path with the minimum cost from A to D (if there is more than one, any one will suffice).

1. \( s = <A> \) \[ S = \begin{array}{c}
\bullet \\
A, 0
\end{array} \]

2. \( S \) is non-empty. The only path in \( S \) is \(<A>\). Extend this and add the extensions (along with their costs) to \( S \):

\[ S = \begin{array}{c}
\bullet \\
A \\
\bullet \\
C, 1 \\
\bullet \\
E, 2 \\
\bullet \\
B, 4
\end{array} \]
Example (continued)

3. \( S \) is non-empty. Select the minimum-cost path from \( S \), which is \( <A, C> \), which has a cost of 1. There are edges from \( C \) to \( A, E \) and \( D \), but we’ve already visited \( A \), so the only extensions we’ll add are to \( E \) and \( D \).

4. \( S \) is non-empty. Select the minimum-cost path again; this time it is \( <A, E> \), and add its extensions.

5. \( S \) is non-empty. For our minimum-cost path, we could select either \( <A, C, E> \) or \( <A, B> \). For the sake of adventure, let’s extend \( <A, C, E> \).
Example (continued)

6. $S$ is non-empty. We select $<A, B>$ as our minimum-cost path, and extend it.

7. Now we could either select $<A, E, C>$ or $<A, B, D>$, which have equal costs. Let’s select $<A, B, D>$. This ends with $D$, and has a cost less than equal to any of the other paths in $S$. Therefore, this is the minimum path to $D$; there might be other paths to $D$ (and in fact, we’ve already discovered two: $<A, C, E, D>$ and $<A, C, D>$, but they must cost at least as much as this path.

So we are finished, and a minimum cost path is $<A, B, D>$ with cost 5.

**Note** (yet again) the search does not end when we find a path to $D$. It continues until we find a path to $D$ that is less expensive than any other path under consideration.
Minimum Paths and Negative Costs

- If the network contains edges that have a negative cost, Dijkstra’s algorithm (as we have seen it) falls over with a dull thud. It assumes that the cost of a path always increases (or perhaps stays the same) as edges are added - so we should always extend the cheapest path, and once we have found the first path, no cheaper paths can possibly remain undiscovered. These assumptions are false if the edges can have a negative cost.

- It’s even uglier if the graph contains a negative cycle because then it is never in the searching algorithm’s best interest to get out of the cycle - the minimum cost is negative infinity, achieved by traversing the negative cycle an infinite number of times.

An aside: as mentioned in Weiss, Dijkstra’s algorithm is an example of a greedy algorithm. A greedy algorithm always does what appears to be the best action based on the current situation, always assuming that this will not cause a problem later. In many situations, greedy algorithms behave optimally. In other situations (such as we saw in the sections on amortized analysis of memory allocation and splay trees), “investing” work in a non-greedy manner is better.
An Approach that Will Not Work

There is a temptation to solve this problem by finding the edge with the minimum cost $M$ and then adding $M$ to the cost of every edge in the network. The resulting graph has only zero or positively costed edges, so it will work with Dijkstra’s Algorithm, but the answer might be wrong!

- The minimum path from A to B is the path $<A,X,Y,Z,B>$, not $<A,W,B>$.
The Minimum Spanning Tree

• We can compute the spanning tree of an undirected network using the same method as an undirected graph. However, with networks there is another question we can ask:

    Since a network can have many possible spanning trees, how can we find a spanning tree that has the lowest total cost?

Such a spanning tree is called a *minimum spanning tree*.

• We can extend Dijkstra’s algorithm for finding a minimum cost path in much the same way as we extended our general path finding algorithm to create a spanning-tree-finding algorithm.

    The main difference is that (as in our spanning tree algorithm earlier) we never add paths that end in a vertex is already present in a path in S.

• The resulting algorithm is called *Prim’s algorithm*. (Described in Weiss).

• Again, note that the minimum spanning tree is not necessarily unique (for example, if all edges in the network have the same cost, *every* spanning tree is minimal).
Kruskal’s Algorithm

• Kruskal’s Algorithm is another method of finding a minimum spanning tree. It is unlike any of the graph algorithms we have seen. (In fact, it does not even employ a traversal)

Kruskal’s Algorithm

1. Let \( T \) be the set of edges in the spanning tree. \( T \) is initially empty.

2. Until \(|V| - 1\) edges have been added to \( T \):
   a. Find the edge \( e \) that
      • has the minimum cost
      • connects two vertices that are not already connected by \( T \).

      **Note:** we don’t know how to do this particularly efficiently because we didn’t study the Set ADT and how to implement this kind of SET operation. If you are interested, consult Weiss.

      b. Add \( e \) to \( T \).

• Kruskal’s algorithm only considers each edge once. Therefore, it doesn’t get baffled by negative edge costs.

• It also works properly for connected directed graphs (although we *may* need to add more than \(|V| - 1\) edges - enough until all vertices are connected).
Kruskal’s Algorithm - Example

1. The cheapest edges are between (A,C) and (B,D). We’ll arbitrarily choose (A,C).

\[ T = \{(A,C)\} \]
Kruskal’s Algorithm (continued)

2. The next cheapest edge is (B,C) (cost 1). Since this edge doesn’t connect two vertices that are already connected by T, add it in.

3. The next cheapest is (A,E) (cost 2). Again, this adds connections, so it’s okay to add to T.
Kruskal’s Algorithm (continued)

4. The next cheapest edge is (C,E) (cost 3), but since C and E are already connected, we keep looking. (A,B) is the next candidate (cost 4), and it does add a connection, so add it to T.

\[ T = \{(A,C), (B,D), (A,E), (A,B)\} \]

- Now T has 4 edges, but no cycles, so T is a spanning tree.
- Because of the manner that we selected the edges, it is also a minimum spanning tree.
Why does Kruskal’s Algorithm Work?

A key observation about Kruskal’s algorithm is that unlike the traversal-based algorithms we have seen, it does not build a set of paths starting at one vertex (although for particular networks, it may appear to be doing so). Instead, it constructs a forest of minimum spanning trees for disjoint subsets of $|V|$, merging existing spanning trees until finally only one spanning tree with $|V| - 1$ edges remains in the forest.

We can prove that Kruskal’s algorithm is correct using an inductive argument. (A full proof can be found in Lewis and Denenberg.)

Base case: the forest consists of all the vertices. Each tree in the forest is a minimum spanning tree for the graph consisting of that single vertex.

Inductive case: we proceed by joining adjacent trees (a tree is adjacent to another if there is an edge between a vertex in one tree and a vertex in the second). We always choose the least expensive edge that will merge two adjacent trees, and therefore this will always be the least expensive way to merge these two trees, resulting in a minimum spanning tree for these vertices.

Imagine that this was not the case - then we could remove an edge from the spanning tree and add some other edge to reconnect the pieces of the tree. But any such edge must be at least as expensive as the one we removed - this can’t possibly improve matters.
Graph Isomorphisms

Two graphs $G_1$ and $G_2$ are *isomorphic* if one of the graphs can be made identical to the other by relabelling its vertices.

The relabelling map from the vertices of one graph to the other is called an isomorphism.

For example:

These two graphs are isomorphic. One possible isomorphism from $G_1$ to $G_2$ is $A\rightarrow X$, $B\rightarrow Z$, $C\rightarrow W$, $D\rightarrow Y$. 
Hard to Find, Easy to Check

For the previous example, it is easy to see that the graphs are isomorphic, and what one isomorphism is. In general, however, determining whether two large graphs are isomorphic and finding an isomorphism can be very difficult. The precise complexity for all graphs is unknown, but there are many kinds of graphs for which the best known algorithms for finding isomorphisms is very expensive (and it is suspected that no fast algorithms exist for many kinds of graphs, such as graphs with the same number of edges for each vertex).

However, given a mapping from vertices to vertices, we can easily check whether the mapping is an isomorphism: just perform the mapping and then compare the edge sets.

So, we have a problem that is very hard to solve, but the answer is very easy to check for correctness.

This sort of problem, although seemingly annoying, is actually very useful as a means of authentication in distributed and insecure systems (i.e. the real world, or the Internet).
Introducing Alice, Bob, Eve, and Mallory

Before we proceed, we will need to learn some of the jargon used to discuss cryptography.

• A protocol is a known set of rules and procedures (i.e. algorithms) for how communication takes place. All of the entities involved in the communication know the protocol.

• The entities in a communications protocol are usually given a standard set of names to indicate their roles:
  
  • When the protocol is between two entities, they are given the names Alice and Bob. The protocol is always initiated by Alice.
  
  • Eve is an eavesdropper. Her goal is to learn as much as possible about what is communicated between Alice and Bob. Ideally, she learns everything that Alice and Bob learn. We assume that she hears everything Alice and Bob say.
  
  • Mallory is malicious. He will do anything he can to disrupt the communications and prevent Alice and Bob from communicating.
  
  • Eve and Mallory may work together; Mallory might disrupt the communication in a manner that makes it easier for Eve to listen in, or Eve might tell Mallory something that she overheard that makes it easier for him to disrupt Alice and Bob.
Authentication Protocols

An authentication protocol is a protocol that can be used to prove the identity of one of the entities to the other.

For example, typing your password when you log in is part of an authentication protocol - presumably you are the only person who knows what your password is, so you are the only person who can correctly answer the challenge when asked for your password.

However... what if Eve is listening? If she is able to observe your password, she now knows everything she needs to log in as you! On a network, your password may pass any number of Eves on its way from your keyboard to a remote host. This is not a secure system, unless something is done to disguise the password before it is sent over the network.

Graph isomorphisms can be used to construct authentication protocols that are safe from Eve.

Before the protocol begins, Alice selects a graph $G_0$ and then creates $F_A$, a random isomorphism of $G_0$. When applied to $G_0$, it creates graph $G_1$. She shares $G_0$ and $F_A$ with Bob via a secure channel that Eve cannot possibly listen in on. (How is that done? That is another story...)

From now on, we assume that Eve can listen to everything Alice and Bob say...
The Authentication Protocol

Alice contacts Bob to discuss the questions on the final exam. Bob would like to make sure that it really is Alice he’s talking to (and not Mallory, attempting to put his own problems on the exam, or find out what the exam will be, for his own nefarious purposes).

1. The purported Alice creates a random isomorphism $F_{0T}$ of $G_0$, and applies it to $G_0$, creating $G_T$. She sends $G_T$ to Bob, but does not reveal $F_{0T}$.

   The true Alice knows $F_A$, and $F_{0T}$, so she can construct an isomorphism $F_{T1}$ from $G_T$ to $G_1$.

2. Bob flips a coin. If heads, he asks Alice to reveal $F_{0T}$, and then he checks whether $F_{0T}$ maps $G_0$ to $G_T$. If tails, he asks Alice to reveal $F_{T1}$, and then he checks whether $F_{T1}$ maps $G_T$ to $G_1$. If the maps are not correct, than he isn’t talking to Alice!

3. If the maps were correct, then he might be talking to Alice, or he might be talking to an imposter - but an imposter can only fool Bob about half of the time, so repeat this protocol until Bob is confident that it is really Alice.
Discussion

Note that Bob only asks Alice to reveal either $F_{0T}$ or $F_{T1}$, but never both. The combination of these is $F_A$, so if both are revealed, Eve can easily deduce $F_A$. However, just learning one of these isomorphisms does not make it any easier to find the other.

Note that Alice doesn’t know ahead of time which Note that Bob must be careful that Alice doesn’t keep selecting the same $F_T$ over and over. If she does, then Bob will eventually will have asked for $F_{0T}$ and $F_{T1}$, revealing $F_A$ to Eve.

An imposter Alice who has learned $G_1$ can fool Bob half of the time by choosing a random map, applying it to $G_1$ (instead of $G_0$) and then hoping that Bob will ask for $F_{1T}$, which she knows. Or, she can play by the rules and hope that Bob asks for $F_{0T}$. However, she can’t know both $F_{0T}$ and $F_{T1}$ unless she knows $F_A$, so she will only fool Bob half of the time this way.

Mallory can disrupt things if he is able to forge messages from Alice, making it appear that she is giving the wrong answers - but there’s nothing he can do to learn the secret or impersonate Alice.

If you find this sort of thing interesting, take a look at *Applied Cryptography*, by Bruce Schneier.