Guide to Reductions

Thanks to Julie Tibshirani for helping with this handout!

We'll design many algorithms over the course of this class. Often, the algorithms we'll design and analyze will solve mathematical problems (searching or sorting lists, determining reachability or shortest paths in graphs, etc.). Although these problems are abstract, this very abstraction makes them applicable in a variety of settings.

In many cases, you will come across an problem that is similar in structure to a problem you already know how to solve. When this happens, you can often use an existing algorithm as a subroutine to solve the new problem. This technique is called a *reduction* and is one of the easiest ways to design algorithms.

This handout discusses general advice about how to structure reductions and gives several examples about how to design and analyze reductions.

Designing an Algorithm

As an example of a reduction, let's consider the following problem:

> You want to plug your computer into a projector. Your computer has a VGA output port, but the projector doesn't have a VGA input port. Instead, it has an input port of type X.
>
> You have a box of connectors. Each connector has an input port of one type and an output port of one type (which, in principle, could be the same type). You want to determine whether it is possible to chain some number of connectors together so that you can plug your computer into the projector.
>
> Design an O(n)-time algorithm that determines whether it is possible to connect your computer to the projector, where n is the number of connectors in the box.

How might we solve this problem? The initial description of the problem doesn't immediately look like anything we've seen before – it asks how to chain different types of items together, which isn't a problem we've tried solving yet.

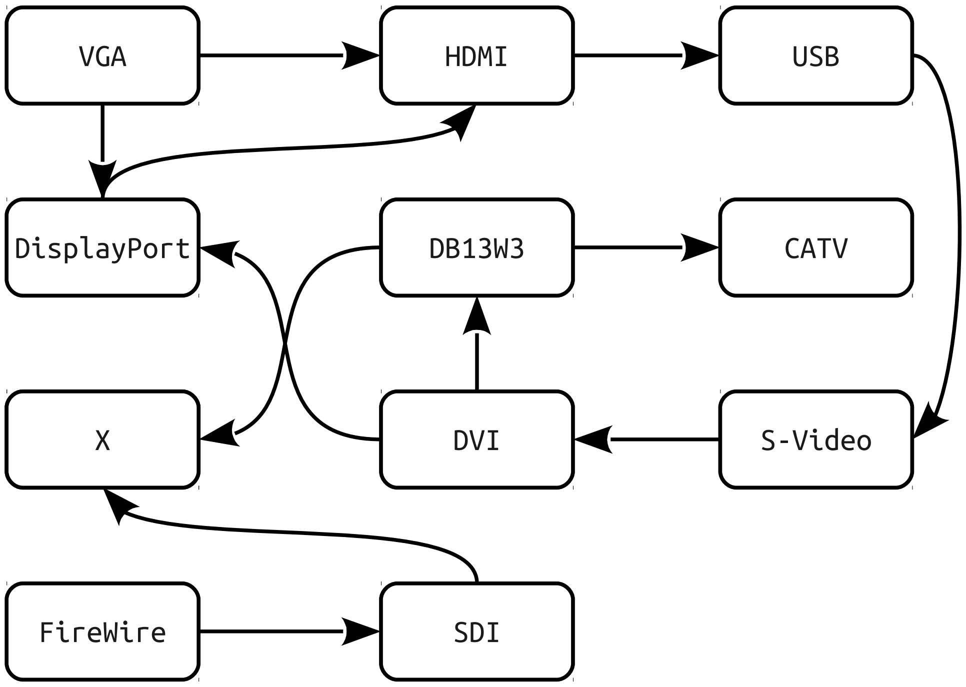
However, you might notice the following. You start off with an output port of type VGA. If you apply a connector with a VGA input port, you will end up with an output port of some new type (let's call it type Y). If you have a connector with a Y input port and a Z output port, then you can chain that onto the end of your connector stack, and now you have a Z output port.

More generally, we can think of the problem this way. At each point in time, there is some “state” the algorithm keeps track of that corresponds to what type of output port is currently at the end of the chain of connectors. The goal is to get to a point where you have an output port of type X, given that you're starting with an output port of type VGA. This is starting to sound a little bit like a graph problem: the nodes correspond to the different types of output connectors that are available, and the goal is to find a path through that graph that starts at VGA and ends at X.

To make this a little more concrete: let's suppose that you have connectors of the following types, where (A, B) denotes that the connector has an input of type A and an output of type B:

(HDMI, USB)	(DB13W3, X)	(VGA, HDMI)
(VGA, DisplayPort)	(DVI, DB13W3)	(DVI, DisplayPort)
(DB13W3, CATV)	(S-Video, DVI)	(USB, S-Video)
(DisplayPort, HDMI)	(FireWire, SDI)	(SDI, X)

We can imagine building up a graph where each node is a connector type and where any connector of the form (*A*, *B*) is represented by an edge from *A* to *B*:



Now that we have this graph, it's a lot clearer how we might go about solving the problem – we just need to check whether there is a path from VGA to X! Indeed, there is: plug the VGA output into the VGA-to-HDMI converter, then plug that into the HDMI-to-USB converter, then plug that into the USB-to-S-Video converter, plug that into the S-Video-to-DVI converter, plug that into the DVI-to-DB13W3 converter, and plug that into the DB13W3-to-X converter. (Phew!)

This observation means that we can solve this problem by modeling it as a graph and then checking whether one node in the graph is reachable from the other. Specifically, we just build the above graph, then return whether VGA can reach X.

Describing and Analyzing the Algorithm

So how exactly would we describe and analyze such an algorithm?

At a high level, you can think of our algorithm as working as follows:

- Construct a graph *G* based on our data.
- Determine whether X is reachable from VGA.

If you were writing up this algorithm for a problem set, you could do so quite simply as follows:

Algorithm: Create a new graph *G* with one node per type of connection mentioned in the list of connectors, plus a node for X and a node for VGA if they don't already exist. For each connector (*A*, *B*), add an edge from *A* to *B* in the graph. Then, return whether X is reachable from VGA in this graph.

This algorithm description might not at all seem satisfactory – we're glossing over a lot of details about how you would build the graph or how you would check if X is reachable from VGA. For the purposes of this class, it's perfectly fine to gloss over these details now. Those are mostly implementation details. Later on, when we discuss the runtime of the algorithm, we'll fill in the details so that we can more precisely determine the work done. This makes the details of *what* the algorithm is doing a lot more clear, which simplifies the correctness proof.

Useful Tip #1: When doing a reduction, it's useful to give the algorithm at a high-level and describe implementation details in the runtime analysis.

Now that we have the algorithm, how would we prove it's correct? Let's think about how the algorithm works. We start off by building a graph from our data. Once we have the graph, we run a known graph algorithm on it (reachability). When we do this, it's critically important to remember that the reachability algorithm has *no conception whatsoever* of display ports, connectors, or any of the specifics of our problem. It just knows about nodes and edges. Consequently, the majority of the work in the proof will be showing why running a reachability algorithm in the graph we build has any bearing whatsoever on the problem at hand.

To prove correctness, we ultimately need to show the following:

Our algorithm returns true iff there is a series of connectors that turns VGA into X

Why do we need a biconditional here? Hopefully the following will clarify a bit. Consider the following algorithms:

Algorithm 1: Return true.

Algorithm 2: Return false.

These algorithms clearly don't solve the problem at hand. However, we can prove the following statements about each:

If there is a series of connectors that turns VGA into X, then Algorithm 1 returns true.

If Algorithm 2 returns true, there is a series of connectors that turns VGA into X.

Notice that each of these statements is just half of the above biconditional. The fact that these statements are true is pretty much meaningless, since the algorithms obviously don't work in all cases! The reason we need to prove the biconditional statement is to ensure that if the answer is "yes," our algorithm returns true, and if the answer is "no," our algorithm returns false. Only by showing that both of these directions hold true can we conclude that our algorithm works correctly.

Useful Tip #2: If an algorithm tests for some property, a correctness proof should show that the algorithm returns true iff the property exists.

So let's get back to our (correct) algorithm. How would we go about establishing that the algorithm is correct? One way to do this would be to prove this statement:

There is a path from VGA to X in graph G
(★) *iff*
There is a series of connectors that turns VGA into X

Notice that this statement is a biconditional – we need to show both that if there is a series of connectors that turns VGA into X, then there is a path in the graph and vice-versa. Why is this? Well, ultimately we want to prove

Our algorithm returns true iff there is a series of connectors that turns VGA into X

By construction, we have that our algorithm returns true iff there is a path from VGA to X in the graph *G*. Consequently, if we can show (★) is true, then we can conclude that the algorithm returns true iff there is a series of connectors that turns VGA into X.

So how exactly would we prove that (★) is true? Since it's a biconditional, one simple approach would be to prove both directions of implication independently. Although in some cases you can directly prove the biconditional, this often ends up being much harder than just showing both directions of implication. Accordingly, the following proof shows that both directions hold:

Theorem: There is a path from VGA to X in *G* iff there is a series of connectors that turns VGA into X.

Proof: We prove both directions of implication.

(⇒) Suppose there is a path from VGA to X in *G*. Since a path exists, there must be a *simple* path (i.e. a path that does not repeat any nodes or edges) from VGA to X; let this path be $P = v_1, v_2, \ldots, v_k$ where v_1 = VGA and v_k = X. Since there is an edge from v_1 to v_2, by construction there must be a connector that takes connection type v_1 as input and produces connection type v_2 as output; the same holds more generally for any v_i, v_{i+1}. Consequently, there is a series of connectors $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$ that begins with VGA and ends with X. Therefore, there is a series of connectors that turns VGA into X.

(⇐) Suppose there is a series of connectors $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$ that begins with VGA and ends with X. By construction, each connector of the form (v_i, v_{i+1}) gives rise to an edge (v_i, v_{i+1}) in *G*. Consequently, $v_1, v_2, \ldots, v_k$ gives a path in *G* that starts with VGA and ends with X, as required. ■

Now that we know that our algorithm is correct, let's analyze the runtime. Given just the high-level description of the algorithm from earlier, there really isn't a simple way to do this. However, we can start to fill in the details. We'd probably build the graph by starting with an empty adjacency list and determining how many types of connectors there are. From there, we can start to add edges in the adjacency list however we'd like. We can then determine reachability by running either BFS or DFS from VGA to X.

Here's a sample runtime analysis that we consider to be at an appropriate level of detail:

Runtime: The graph we will be building has O(n) nodes and O(n) edges, since each connector contributes at most two nodes and one edge. We can construct each node and each edge in O(1) time each, so the total time spent constructing the graph is O(n).

Once we have the adjacency list, we can determine whether VGA can reach X by running a DFS starting at X and seeing whether we ever reach VGA. This will take time O(n), because there are O(n) nodes and edges.

Overall, this algorithm takes time O(n).

Two Practice Problems

Here are two sample problems to help you practice designing, analyzing, and proving correctness of algorithms that use reductions. Solutions are on the following pages.

Problem One: Nearby Points

Suppose that you have a collection of n locations in 2D space expressed as (x, y) pairs. You are interested in determining the k points closest to your current location p. Design an $O(n)$-time algorithm for finding the k points closest to p. For simplicity, you can assume that no two points are at the same distance to p.

Problem Two: Reliable Routing

Suppose that you have a collection of Internet routers, some of which are directly connected to one another by links. When transmitting a message across a link, there is some probability that the message will fail to arrive intact. Different links have different probabilities of success; specifically, for each link between two computers A and B, there is a success probability $0 < p(A, B) \leq 1$ associated with the link from A to B.

Design an $O((m + n) \log n)$-time algorithm for finding the path from some computer S to a destination computer T that has the highest probability of success, where n is the number of computers and m is the number of links.

Solution to Nearby Points

Algorithm: For each point x, compute the distance d from x to p and append (x, d) to an array A. Then, use the linear-time selection algorithm to compute the kth order statistic in this array, using the distance values as the criterion on which to sort. Finally, return the points stored in the first k positions in the array.

Correctness: If we were to sort the array A into ascending order by distance, we would have a list of all nodes sorted by their distance to p. The first k of these would then consist of the k points closest to p.

When the linear-time selection algorithm runs, it will reorder the array so that the element at position k is the element that would appear at position k in sorted order and all elements that precede it in the sorted order would appear before position k. Consequently, when the algorithm returns the first k elements of the array after running the linear-time selection algorithm, it returns the k elements closest to p. ■

Runtime: Our algorithm does $O(n)$ work computing the distance from each point to p when constructing the initial array, then does $O(n)$ work running the linear-time selection algorithm. It then does $O(k)$ work returning the first k elements of the array, and since $k = O(n)$, the total work done is therefore $O(n)$.

Solution to Reliable Routing

Algorithm: Our algorithm is as follows. Create a graph G with one node per computer. For each link from computer A to computer B with reliability $p(A, B)$, create an edge from A to B with associated weight $-\log p(A, B)$. Then, find a shortest path from S to T and return that path.

Correctness:

Lemma: There is an S-T path of length l in G iff there is an S-T path of reliability e^{-l} in the network.

Proof: Consider any S-T path in G of length l. This path must have the form $v_1, v_2, \ldots, v_k$, where $S = v_1$ and $T = v_k$. The length of this path is then given by

$$l=\sum_{i=1}^{k-1} l(v_i, v_{i+1})=\sum_{i=1}^{k-1} -\log p(v_i, v_{i+1})=-\sum_{i=1}^{k-1} \log p(v_i, v_{i+1})=-\log\prod_{i=1}^{k-1} p(v_i, v_{i+1})$$

Note that by construction of G, there is an edge (v_i, v_{i+1}) iff there is a link from computer v_i to computer v_{i+1}. Consequently, there is a path $v_1, v_2, \ldots, v_k$ in G iff there is a corresponding series of links connecting computers S and T. Moreover, the probability p that a message sent from S to T along that path will reach its destination is given by the product of all the individual probabilities along that path, which is

$$p=\prod_{i=1}^{k-1} p(v_i, v_{i+1})$$

Consequently, we have that $e^{-l} = p$, so this path also has reliability e^{-l}. Therefore, there is a path from S to T in G of length l iff there is a corresponding transmission path from S to T with success probability e^{-l}. ■

Theorem: The algorithm correctly finds a maximum-reliability path from S to T.

Proof: First, note that the notion of a maximum-reliability path is well-defined, since if we consider any path of success probability p, adding any additional edges onto that path can only decrease the success probability.

Next, suppose our algorithm returns a path whose length in G is l; this must be a minimum-cost path from S to T. There is therefore a corresponding path from S to T with reliability e^{-l}. Since e^{-l} is a monotonically decreasing function of l, it is maximized when l is minimized. Since our algorithm returns a path whose l value is minimized, this means that the path it returns maximizes the success probability. ■

Runtime:

We can construct the graph G in time $O(m + n)$ by adding in one node per computer and one edge for each link; each can be built in $O(1)$ time.

We claim that we can run Dijkstra's algorithm in the resulting graph to find shortest paths. Since our graph has n nodes and m edges, running Dijkstra's algorithm will take time $O((m + n) \log n)$, which, combined with the $O(m + n)$ time to build the graph, yields a runtime of $O((m + n) \log n)$. To see this, note that all edges have nonnegative weights; since $0 < p(A, B) \leq 1$, we know that $-\infty < \log p(A, B) \leq 0$, so $0 \leq -\log p(A, B) < \infty$.