Polynomial Reductions

Fortunately, most of the reductions that we do in practice are far less arduous than the CNF-SAT to k-Clique reduction. For most new problems we are likely to encounter, the odds are very high that someone has already proved that a similar problem is NP-Complete. Reductions between similar problems are usually pretty straightforward.

There are some well-established techniques for creating reductions - we will look at a couple here.

We have already defined these two problems:

**Partition**: Given a set of integers, is it possible to divide the set into two subsets so that each subset sums to exactly 1/2 the sum of the original set?

**Subset Sum**: Given a set of integers and a target integer k, does the set have a subset that sums exactly to k?

Both of these are obviously decision problems, and both are clearly in NP (Situation check - make sure you understand the difference between saying a problem is in NP, and saying the problem is NP-Complete. This distinction is essential.)

Let's suppose for the moment that we know that **Partition** is NP-Complete (it is), and see how we could use that knowledge to prove that **Subset Sum** is NP-Complete. It turns out that this is extremely easy because the new problem (SS) is a **generalization** of the known problem (Part). Or putting it another way, Partition is a special case of Subset Sum. Transforming an instance of Partition into an instance of Subset in an answer-preserving form takes no effort at all - every instance of Partition is already an instance of Subset Sum.

This illustrates a general principle: **if a new problem (in the class NP) is a
generalization of a known NP-Complete problem, then the new problem is also NP-Complete.

Now let's suppose that we know Subset Sum is NP-Complete, but we don't know about Partition. Can we use Subset Sum to show that Partition is NP-Complete? Well, we can't use the principle we just learned - we can't automatically transform an instance of a general problem (SS) to a special case (Part).

We can still find a reduction ... it just takes a little bit more work.

Let's think about what we start with: an instance of Subset Sum, i.e. a set of integers S and a target integer k. We want to transform this into an instance of Partition in such a way that the Yes/No answer is preserved. Suppose the answer to the instance of SS is Yes - that means there IS a subset that sums to k. If we let T be the total sum of the set, then the other subset (i.e. all the elements not in the set that sums to k) must sum to T-k. We need to "do something" to the original set of integers S so that we have a set that can be partitioned into two equal sets.

It turns out that this is very easy. Consider modifying S by adding two new very large values to it ... call these new values X and Y, and call the modified set S'. Treat this as an instance of Partition. If the answer to this Partition instance is Yes, then S' can be split into two subsets with equal sums. Because X and Y are both very large, the new value X is in one of these subsets, and the new value Y is in the other. Can we choose X and Y so that the other values in X's partition sum to k?

Let P_X be the partition that contains X, and P_Y the partition that contains Y.

We need \( \sum_{P_X} = X + k = \sum_{P_Y} = Y + T - k \)

and from this we can see that we need \( Y = X + 2k - T \)

Even requiring that X and Y are very large with respect to the other values, we
see that there are infinitely many X, Y pairs that satisfy this. In class we used
\[ X = 100T - k \text{ and } Y = 99T + k, \]
but we could just as well have used \[ X = 100T \text{ and } Y = 99T + 2k \text{ etc.} \]

With X and Y defined and added to S to give S', we can show that S' has a partition if and only if S has a subset that sums to k.

We need to formalize the transformation and show that it is answer-preserving.

**Transformation:** Given instance (S,k) of Subset Sum,
- compute \( T = \text{total sum of } S \)
- compute \( X = 100T - k \) and \( Y = 99T + k \)
- create \( S' = S + \{X, Y\} \) (i.e. add the element X and Y to S)
  (note that the total sum of \( S' \) is 200*T)

This transformation clearly requires no more than \( O(n) \) time, where n is the size of S.

To show the transformation is answer-preserving:

1. Suppose the answer to Subset Sum(S,k) is Yes. Then S contains a subset Q that sums exactly to k. Q is also a subset of S'. Thus including X in Q gives a subset of S' that sums to 100*T. The other elements of S' must also sum to 100*T (because the total sum of S' is 200*T), so the answer to Partition(S') is Yes.

2. Suppose the answer to Partition(S') is Yes. Then S' can be split into two subsets, each summing to 100*T (because the total sum of S' is 200*T). X and Y cannot both be in the same subset, so X must be in a subset with values from S that sum to k. Thus the answer to Subset Sum(S,k) is Yes.

Therefore Subset Sum reduces to Partition.

This is an example of transforming a general case into a specific case - we can't always do this but for these two problems it is simple because they are so
So far we have discussed reduction as a process of transforming an instance of one problem into an instance of another problem. Sometimes we can show that a new problem is NP-complete by giving a transformation that turns any instance of a known NP-complete problem into several instances of the new problem, where a Yes answer in any of the instances of the new problem corresponds to a Yes answer in the original instance of the known problem.

For example, consider the following problems:

Recall that the length of a path in a graph is the number of edges it contains, not the number of vertices.

P1: Given a graph G on n vertices, does G contain a path of length n-1? (This is the Hamilton Path problem).

P2: Given a graph G on n vertices, and two specified vertices A and B in G, does G contain a path with length n-1 with ends at A and B?

Problem P2 is identical to problem P1 except that the ends of the path have been specified. We might guess that this makes the problem easier, but in fact we can show that P1 reduces to P2:

Let G be an instance of P1. Let the vertices of G be \{v_1, v_2, v_3, ..., v_n\}

Create instances of P1 of the form (G, v_i, v_j) where i and j both range from 1 to n, with i != j. Note that there are O(n^2) such instances.

If the answer to P1(G) is Yes, then at least one of the instances of P2(G, v_i, v_j) will have answer Yes. Similarly, if any of the instances of P2(G, v_i, v_j) has answer Yes, then P1(G) has answer Yes.

Since creating each instance of P2 takes polynomial time, and there are a
polynomial number of instances, the total amount of work done in the transformation requires polynomial time. Since the transformation preserves Yes answers (albeit in a slightly different way than before) this is a valid reduction.

Since problem P1 is known to be NP-complete, we can conclude that P2 is also NP-complete.

This technique is called one-to-many reduction, and is valid as long as we can be sure that only a polynomial amount of work is done.

The bad news about NP-Complete problems is that nobody has ever found an algorithm that solves any of them for all cases in less than exponential time. Later in the course we will look at algorithms that solve some cases of some NP-Complete problems quite efficiently, but we can't get away from the worst-case exponential time requirement for finding the solution to some other instances of the problems.

That being said, we may be able to reduce the time requirement for solving an NP-Complete problem, even though it still remains exponential.

For example, consider (once again) the Subset Sum problem: Given a set S of n integers and a target value k, does S have a subset that sums to k?

The naive ("brute force and ignorance") algorithm simply examines every subset of S to see if any of them sum to the target value k. Since S has $2^n$ subsets, this algorithm runs in $O(2^n)$ time. (You may wonder why I don't include a time factor for computing the sum of each subset - in fact, the sum of each subset can be computed in constant time. See if you can see how to do this.)
We can improve on the BFI algorithm as follows. This very clever method was first described by Horowitz and Sahni.

1. Split S arbitrarily into two equal sized subsets S1 and S2. If S has an odd number of elements, make the split as even as possible. It doesn't matter which of S1 or S2 is bigger in this case.

   # If S does have a subset that sums to k, there are three possibilities:
   # - all required elements of S are in S1
   # - all required elements of S are in S2
   # - some required elements of S are in S1, and some are in S2

2. Compute Sums1 = {sums of all subsets of S1}
   Compute Sums2 = {sums of all subsets of S2}

3. If k $\in$ Sums1 or k $\in$ Sums2:
   report "Yes" and stop # this takes care of the first two possibilities

4. Else:
   # we need to determine if there is a subset of S1 that can be combined with a subset of S2 to give a sum of k.
   # This is equivalent to asking if there is an x in Sums1 and a y in Sums2 such that x+y = k

5. Sort Sums1 into ascending order
   - label the elements x1, x2, ...

6. Sort Sums2 into ascending order
   - label the elements y1, y2 ...

7. Let i = 1 and let j = size(Sums2)

8. While i $\leq$ size(Sums1) and j $\geq$ 1:
9. \[ z = x_i + y_j \]

10. If \( z = k \):
    
    report "Yes" and stop

11. Elsif \( z < k \):
    
    # this implies that \( x_i + y_h < k \ \ \forall h < j \),
    # i.e. \( x_i \) is too small to be in any
    # solution to the problem

12. \( i += 1 \) # move to the next value in Sums1

13. Else: # \( z > k \)
    
    # this implies that \( x_h + y_j > k \ \ \forall h > i \),
    # i.e. \( y_j \) is too big to be in any solution

14. \( j -= 1 \) # move to the previous value in
    # Sums2

    # exit the While loop because we have exhausted the possibilities

15. report "No"

You should convince yourself that this algorithm correctly solves Subset Sum. We now determine its complexity.

Step 1 is trivial. Step 2 takes \( O(2^{(n/2)}) \) time since each of S1 and S2 has n/2 elements. Sums1 and Sums2 each have \( 2^{(n/2)} \) elements. Sorting each of Sums1 and Sums2 takes \( O(2^{(n/2)} * \log(2^{(n/2)})) \) time, which simplifies to \( O(n * 2^{(n/2)}) \).

The loop in lines 8 .. 15 iterates at most \( 2 * 2^{(n/2)} \) times, doing constant-time work each time.

Thus the dominant step is sorting, and the entire algorithm runs in \( O(n * 2^{(n/2)}) \)
time.

This is still exponential but it is **way better** than the BFI algorithm.
Divide and Conquer Algorithms

The Divide and Conquer Paradigm

To solve a problem of size n:
   If n is "small":
     solve the problem directly
   else:
     Subdivide the problem into two or more (usually disjoint) subproblems
     Solve each of the subproblems recursively
     Combine the subproblem solutions to get the solution to the original problem

Examples of D&C algorithms are familiar to everyone who has studied computing: binary search, Quicksort, Mergesort are classic examples.

In this class we looked at a possibly less-familiar application of D&C: finding the longest path in a tree.

First, let us consider the more general problem: given a graph G, what is the longest path in G? We can call this the max-path problem.

This is the optimization version of a problem that almost made it onto our Magnificent 8 Problems: the Hamiltonian Path Problem, which simply asks the question “Does graph G contain a path that includes every vertex of G?” This problem is NP-Complete and since it reduces to max-path, we conclude that there is almost certainly no polynomial-time algorithm for max-path.

However, the story changes when we restrict the question to trees.

Definition: A tree is a connected graph with no cycles. The practical implication of this is that for any pair of vertices in a tree, there is exactly one
path of edges that joins them.

In a tree, some vertices are joined by short paths and some by longer paths. If the tree represents a message passing network, the length of the longest path is of interest as it gives some kind of measure of the maximum delay for a message to travel from its source to its destination.

Computing the longest path is not difficult. A simple Breadth-First Search or Depth-First Search algorithm can be used n times, to start at each vertex and compute the lengths of the paths from that vertex to all others in O(n) time (this algorithm has a higher complexity on graphs that are not trees). Since in a tree there is exactly one path between each pair of vertices, we don't have to consider alternative paths between two vertices when looking for the longest path: the length of the longest path is simply the maximum distance between any two vertices of the tree. Finding the distances from each vertex to all the others, and keeping track of the longest distance, gives a simple O(n^2) algorithm for finding the longest path in the tree: apply an O(n) search algorithm n times.

But we can do better!

Suppose we have a tree T. Choose an arbitrary vertex x as the root of T - we can visualize grabbing vertex x and pulling it upwards, so that the rest of the tree "hangs down" from x. Let T1, T2, ... represent the subtrees that hang from x.

Now we can make an observation about the longest path in T. It may seem a bit obvious but it is very useful:

Either the longest path in T goes through vertex x ...
... or it doesn't

Trivial as it may seem, this observation motivates our algorithm. Because if the longest path in T doesn't go through x, then it (the path) must be completely contained in one of the Ti subtrees ... and we can find the longest path in each subtree recursively (see how craftily Divide and Conquer creeps in).
However, it is entirely possible that the longest path in T does go through x (remember, we chose x arbitrarily). In this case, the longest path must "come up out of" some Ti, pass through x, then go "down into" some other Tj. And since this path is the longest possible path in T, we must be using the longest path that goes from the bottom to the top of Ti, and the longest path that goes from the bottom to the top of Tj.

So for each Ti, we need the longest path that is completely contained in Ti, and we also need the longest path that goes from the bottom of Ti to the top of Ti. From this info, we can compute the longest path in T.

Algorithm Max_Tree_Path(T, x)  // x is the root of T (T may be a subtree of the original tree)
  # We compute two paths for T:
  #    C : the longest path in T
  #    R : the longest path in T with one end at x
  # This algorithm will return both of these paths - if implemented in a language that permits only single item returns,
  # an object containing both of these paths must be constructed and returned.

  # base case
  if T consists of just the vertex x:
    C = {x}   - the longest path in T
    R = {x}   - the longest path from the bottom to the top of T
  else:
    Let T1, T2, ... Tk be the subtrees that are attached to x, each rooted at the vertex xi that connects directly to x
    # create two indexed lists, CS and RS, which will contain the C paths and R paths for the subtrees as they are
    # recursively constructed
for i = 1, 2, ... k
    CS[i], RS[i] = Max_Tree_Path(Ti, xi)  # this recursive call returns
the C and R paths for subtree Ti

    # from the information returned by the recursive calls on the subtrees,
we compute C and R

    # R is easy
    R = max(RS[i]) + {x}  # here "+" means "append to the
path"

    # C takes a bit more work. The longest path in T may be completely
contained in one of the subtrees, or it may
    # use x as a bridge to get from one subtree to another

    Longest_Subtree_Path = max(CS[i])  # this is the longest path
completely within any of the subtrees
    Longest_Path_Through_x = max(RS[i]) + {x} + second_max(RS[i])

    # second_max is an easily defined function that returns the second
largest of the paths.
    # If there is no second largest, it returns a null value
    C = max(Longest_Subtree_Path, Longest_Path_Through_x)

return C, R  // using a Python-like return of multiple objects

This algorithm has some interesting points. In most D&C algorithms we know
eactly what information we need from each subproblem, and that is what we
compute. In this algorithm we don't know how we will need to combine the
information gained by solving the subproblems to get the overall solution, so we
compute some things that we may not actually use. We compute them because
they may turn out to be useful, but we don't know that in advance.
The complexity of the algorithm may look difficult to compute. We can define \( MTP(T) \) to be the time required for \text{Max_Tree_Path} to run on a tree \( T \), and the recurrence relation would look something like this:
\[
MTP(T) = c + MTP(T_1) + MTP(T_2) + \ldots MTP(T_k)
\]
where \( T_1 \ldots T_k \) are the subtrees, and we can't tell in general how big they will be.

However, if we think of the "execution tree" of the recursive calls made by the algorithm, we see that every single vertex of the tree gets to be the root of a subtree exactly once during the execution of the algorithm.

So the number of calls to the function is exactly equal to the number of vertices, and the work related to each recursive call is constant (just a few comparisons and list concatenations). Thus the algorithm runs in \( O(n) \) time - a full order of magnitude faster than the algorithm that we started with. If we are dealing with a very large tree, that's a big advantage.

The existence of such a fast algorithm for solving on trees a problem which is known to be extremely difficult to solve on general graphs leads to an obvious but profound question: does it only work on trees, or can we use the same technique on other types of graphs? For example, what if we add one edge to a tree - can we still solve the longest-path problem in polynomial time? (The answer is Yes - try to figure out how to do it.) It turns out that the answer is that there are many families of graphs for which this and other NP-Complete problems can be solved in polynomial time. These results are beyond the scope of this course but they are not difficult to understand. For more information search for "k-terminal recursive graphs" and in particular, papers by Steve Hedetniemi and Tom Wimer.