Design and Analysis of Algorithms – CSE 101
Using reductions between problems to design algorithms

1 Reductions

Reducing one problem to another is actually one of the most useful techniques in algorithm design, and one that is done implicitly almost every time we apply an algorithm to a real-life situation. Many textbooks do not address the reduction method explicitly, although most have many problems intended to encourage reductions. This guide is meant to give students more details in how to use reductions, how to prove they are correct, and how to perform a time analysis for the resulting algorithms. Reductions are also used to argue about the computational difficulty of problems, in NP-completeness, and even uncomputability of problems, as in reductions from the halting problem (covered in CSE 105). But here our focus is on using reductions to problems where we know an efficient algorithm.

2 Decision problems:

A decision problem is one where the answer is "Yes" or "No", or "True" or "False". Reductions are used for other types of problems as well, but the situation is simplest for decision problems, and other types of problems can often be phrased as decision problems. (We'll also briefly touch on reductions between search problems below).

If Π is a decision problem, and x is an instance, we often write $x \in \Pi$ to mean the answer to input x is "Yes". (In other words, we identify a decision problem with the set of "Yes" instances.) A reduction between Π_1 and Π_2 is a way to convert instances of Π_1 into equivalent instances of Π_2, or a way of "coding" questions $x_1 \in \Pi_1$? as $x_2 \in \Pi_2$? More precisely, a reduction between decision problem Π_1 and Π_2 is an algorithm F that, given instance x_1 of Π_1, outputs an instance $x_2 = F(x_1)$ of Π_2, so that $x_1 \in \Pi_1$ if and only if $x_2 \in \Pi_2$.

3 Algorithm design

Say we already know an algorithm Alg_2 that solves Π_2. Then we can use a reduction F to construct an algorithm Alg_1 for Π_1 as:

```
$Alg_1(x_1)$
```

1. $x_2 := F(x_1)$
2. Return $Alg_2(x_2)$

3.1 Correctness

To prove correctness of this algorithm, we can use the correctness of Alg_2 as given, so what remains to prove is that F really is a reduction. In other words, we need to assume $x_1 \in \Pi_1$ and show $x_2 \in \Pi_2$, and then assume $x_2 \in \Pi_2$ and show $x_1 \in \Pi_1$,

3.1.1 Existence problems

One common form of decision problem is an existence problem. For example, "Is there some path from s to t in G?". "Is there a triangle in the graph?" "Are there two numbers in this list that add up to a third in the list?" "Is there a list of flights that will take a passenger from city 1 to city 2 before time T?"

Logically, we can express such a problem in the form $x \in \Pi \exists y R(x, y)$, where R is a relation. We read this as "x is a yes instance of Π if there is some solution y that meets the constraints $R(x, y)$.

To show that F is a reduction between Π_1 given as" $\exists y_1 R(x_1, y_1)$" and Π_2 given as " $\exists y_1 R(x_1, y_1)$", we need to show that there is a solution in the sense of Π_1 to x_1 if and only if there is a solution in the sense of Π_2 to x_2.

Drilling down, we need to:

- Define the reduction F, describing x_2 in terms of x_1. We only know x_1 when we define x_2, we cannot use y_1 since we don't even know it exists..
- If $x_1 \in \Pi_1$ then $x_2 \in \Pi_2$.
    1. Assume we have a solution y_1, so that $R_1(x_1, y_1)$.
    2. Use y_1 to define a solution y_2
    3. Prove that y_2 is a valid solution in the sense of Π_2, i.e., that $R_2(x_2, y_2)$. Here, we need to use: Definition of x_2, that $R_1(x_1, y_1)$, and definition of y_2
- If $x_2 \in \Pi_2$ then $x_1 \in \Pi_1$.
    1. Assume we have a solution y_2, so that $R_2(x_2, y_2)$.

2. Use y_2 to define a solution y_1
3. Prove that y_1 is a valid solution in the sense of Π_1, i.e., that $R_1(x_1, y_1)$. Here, we need to use: Definition of x_2, that $R_2(x_2, y_2)$, and definition of y_1

3.1.2 Search problems

A search problem asks you not just to decide whether there is a solution y, but, if there is, to find one. We can use the same method to prove reductions are correct between search problems. The only difference, is that the step where we use y_2 to define a solution y_1 becomes part of the algorithm, so we need to pay attention to the efficiency of this step.

$SearchAlg_1(x_1)$

1. $x_2 := F(x_1)$
2. IF $SearchAlg_2(x_2)$ returns a solution y_2, return the corresponding y_1
3. IF $SearchAlg_2(x_2)$ returns "No solution exists", return "No solution exists"

3.2 Time analysis

We can use the time analysis for Alg_2 as given. But note that the time for step 2 is the time to run Alg_2 on x_2 not on x_1. So the overall process is:

1. Use the definition of F to give an upper bound on all the relevant size parameters (nodes, edges, etc.) for x_2 in terms of those for x_1.
2. Plug these upper bounds in for the size parameters in the time complexity of Alg_2.
3. Add the time to compute F. (Usually, the time for Alg_2 dominates this, but not always)

4 Examples

Let's look at the same examples we considered for path algorithms. Say a graph has edge labels 0 and 1. We want to know if there's a path from s to t with an even number of edges labelled 1. This is the EvenPath problem.

In the max bandwidth path , edges have values $w(e) > 0$ representing the bandwidth of a link, and we want to find the path that maximimizes $F(p) = min_{e \in p} w(e)$. We can define a related decision problem, where we are also given a value B and ask "Is there a path of bandwidth at least B"? Call this problem BandwidthDecision.

We'll reduce both of these to reachability: Is there a path in G from s to t? DFS or BFS solve reachability in time $O(n + m)$.

4.1 Reducing EvenPath to Reachability

Note that we have two ways of visiting a vertex: if the sum of edges along the path so far is even, or if the sum so far is odd. Let's modify the graph to actively keep track of this information. In other words, given labelled graph $G = (V, E), s, t$, we construct a graph G' with vertices $V' = \{(v, 0), (v, 1) | v \in V\}$ that has two copies of every vertex v, one for even paths and the other for odd paths. If an edge $e = (u, v)$ is labelled 0, it does not change the parity bit, so we add two edges $e^0 = ((u, 0), (v, 0))$ and $e^1 = ((u, 1), (v, 1))$ to E'. If e is labelled 1, we add two edges $e^0 = ((u, 0), (v, 1))$ and $e^1 = ((u, 1), (v, 0))$ to E'. The null path is even, so we set $s' = (s, 0)$, and we are trying to reach t by an even path, so we set $t' = (t, 0)$. The claim is that, in G there is a path from s to t with an even number of edges labelled 1, if and only if in $G' = (V', E')$ there is a path from s' to t'.

That is, we reduced the EvenPath problem for G, s, t to Reachability for $F((G, s, t)) = (G', s', t')$

Let p be a path from s to t in G, going through vertices $s = v_o, v_1, ..v_{k-1}, v_k = t$, and let b_i be the sum mod 2 of the labels of the first i edges; we treat $b_0 = 0$. We claim $(s, 0), (v_1, b_1), ...(v_k, b_k)$ is a path in G', because for each $e = (v_i, v_{i+1})$, in G $e^{b_i} = ((v_i, b_i), (v_{i+1}, b_i + l(e) mod 2) = (v_{i+1}, b_{i+1})$ is an edge in G'. If p had an even number of edges labelled 1, $b_k = 0$, so there is a path from $(s, 0)$ to $(v, 0)$ in G'.

In the other direction, if p' is a path from s to t in G' going through $(s, 0), (v_1, B_1), ..(v_k, B_k) = (t, 0)$, $p = s, v_1, ..v_k = t$ is a path in G, and B_i is the sum mod 2 of the first i labels of that path. So p must be a path in G going from s to t with an even number of edges labelled 1.

Thus, G, s, t is a yes instance of EvenPath if and only if G', s', t' is a yes instance of Reachability.

4.2 Time analysis for Even path

We can construct G' by going through every vertex and edge in G, so the time to compute G' is $O(n+m)$. $|V'| = 2n$ since we replace every vertex with 2 vertices, and $|E'| = 2m$ since we replace every edge with two edges. Thus, the time to run graph search on G' is $O(2n+2m) = O(n+m)$. So the overall time is $O(n+m)$.

4.3 Example: BandwidthDecision

A path whose bandwidth is $\geq B$ can only have edges whose weights are $\geq B$. Conversely, if a path has only edges with weight $\geq B$, its minimum edge weight, which is its bandwidth, must be $\geq B$. So if $G = (V, E), s, t, B$ is an input to BandwidthDecision, we can construct the subset of edges $E_B = \{e \in E : w(e) \geq B\}$, and then use DFS to decide whether there is a path from s to t in $G_B = (V, E_B)$. As observed above, if there is a path p in G from s to t whose bandwidth $\geq B$, then the same path is present in G_B. Conversely, if p is a path in G_B, every edge in p has $w(e) \geq B$, so $bw(p) = min_{e \in p} w(e) \geq B$, and the same path is a path of bandwidth $\geq B$ in G. So reachability in G_B is equivalent to being a Yes instance of BandwidthDecision.

Since $E_B \subseteq B$, $|E_B| \leq m$, and we can construct E_B in $O(n+m)$ time by comparing each edge to B, the time for this algorithm is $O(n+m)$.

We can then use *BandwidthDecision* as a subroutine in computing the MaxBandwidth, the largest B so that *BandwidthDecision* is true for G, s, t, B. Sort all of the edge weights, and store them in an array. (This takes time $O(m \log m)$.) Then perform a binary search on this array to find the largest value so that *BandwidthDecision* is true for that value. This will be the largest possible bandwidth. We need $O(\log m) = O(\log n)$ runs of *BandwidthDecision* for the binary search. So the total time will be $O(m \log n + (m+n) \log n = O((m+n) \log n)$.