CS 577: Introduction to Algorithms Fall 2020

How to Develop Reductions

Instructor: Dieter van Melkebeek TA: Ryan Moreno

This handout is meant to provide a basic structure to developing and proving the correctness of reductions. Please note that the write-up steps are not required patterns. Any solution that is clear, correct, sufficiently efficient, and analyzed (correctness and complexity) will get full credit.

Overview and Approach

In a reduction, you have two problems: `ProbA` is the problem that you are trying to solve, and `ProbB` is the problem you will reduce to. We write `ProbA` $\leq^p$ `ProbB` if there is a polynomial time reduction from `ProbA` to `ProbB`. A polynomial time reduction means that you can write an algorithm AlgA to solve `ProbA` that only takes polynomial time if you pretend that you have access to a blackbox AlgB that solves `ProbB` in time $O(1)$. In other words, AlgA can only make a polynomial number of calls to AlgB, and outside of these calls, AlgA can only take polynomial time to create inputs for the call(s) to AlgB and to handle the outputs of the call(s).

For questions asking you to develop a polynomial-time algorithm for `ProbA` using a reduction, you will want to develop a polynomial-time reduction from `ProbA` to `ProbB` (`ProbA` $\leq^p$ `ProbB`) where `ProbB` has a known polynomial-time algorithm. Examples of this type came up in our section on network flow, and there is an example at the end of this handout.

For questions asking you to prove that a problem is NP-hard, you will consider this problem to be `ProbB`. Pay very close attention to the direction you are reducing. In order to prove that `ProbB` is NP-hard, you will take a known NP-hard problem as `ProbA` (e.g., 3-SAT, k-coloring or Independent Set) and you will develop a polynomial time reduction from `ProbA` to `ProbB` (`ProbA` $\leq^p$ `ProbB`). Intuitively, this means that if there was a polynomial time algorithm for `ProbB`, then there would also be a polynomial time algorithm for `ProbA`. But we know that `ProbA` is NP-hard! So `ProbB` must be NP-hard as well.

Suggested Format

1. **Reduction Description**: ∼1 paragraph to explain the reduction (AlgA from above). Pseudocode can be used for clarity, but isn't required. You will need to explain how you build the inputs for the blackbox call(s) to AlgB, and how you handle the outputs of these blackbox call(s) to AlgB in order to develop an output for AlgA.

2. **Correctness**: ∼1-2 paragraphs proving that the output of AlgA is correct by leveraging the correctness of AlgB. See tips for this below.

3. **Runtime Analysis** ∼2-3 sentences explaining why AlgA is a polynomial-time reduction.

Tips for Developing Reductions

- Sometimes we will tell you what problem to reduce to/from. Other times, especially in the section on NP-completeness, you will need to select the problem to reduce to/from. Many times you'll try one problem, get stuck, and need to try another. Just keep trying to make progress where you can.

- When deciding on a problem to reduce to, look for similarities between the problems.

  In the context of reductions to max-flow/min-cut, it can be tempting (and can be helpful) to think about whether `ProbA` is a maximizing or minimizing problem. However, it is even more important to think about which aspects of the problem could correspond to nodes, which to edges, and which to flow or cuts. After you build such a network, you might find yourself needing to know the maximum cut rather than the minimum cut. This is not necessarily a problem, as you can often change the question from maximizing the amount of value picked to minimizing the amount of value not picked.

  For example, in image segmentation, we developed a graph with nodes representing pixels, where all of the nodes representing foreground pixels would be in S and all of the nodes representing background pixels would be in T. With this in mind, we could first try placing an edge from node i to node t with capacity f_i, the value of placing pixel i in the foreground. This way, if the cut went through this edge, including node i in S, we account for the value of placing node i in the foreground. Unfortunately, this leaves us wanting to find the max-cut since we want to maximize $\sum_F f_i$. Instead, we realized that maximizing the sum of the values of nodes placed in the foreground ($\sum_F f_i$) is equivalent to minimizing the sum of the values of nodes *not* placed in the foreground ($\sum_B f_i$). Then we simply placed an edge with value f_i from s to node i. This way, if i belongs to T (pixel i is placed in the background), we accrue a cost f_i for not placing it in the background. So, our new goal is to minimize this cost.

- When deciding on a problem to reduce from, look for similarities between the problems.

  In the context of NP-completeness, consider the objectives of the problem. If you are looking at a packing problem (problems in which you want to pick as many objects as possible under some constraints), consider reducing from another packing problem (e.g., Independent Set). If you are looking at a covering problem (problems in which you want to pick as few objects as possible under some constraints), consider reducing from another covering problem (e.g., Vertex Cover). Other useful categories include feasibility problems such as 3-SAT (search problems that need to find a single solution given many constraints) and sequencing problems such as Hamiltonian Cycle and Traveling Salesperson.

Tips for Proving the Correctness of Reductions

- Remember that your end goal is always to prove that your AlgA produces the correct output. In some cases, your reduction will involve many calls to the blackbox AlgB. In this case, there is not one particular strategy to help you prove the correctness because it depends on the structure of the blackbox calls. Take HW09 #1 as an example, the problem of reducing 3-SAT search to 3-SAT decision. In this case, we looped over the variables, grounding each (i.e., selecting an assignment of `true` or `false`). Because of this loop structure, it made sense

to prove that at each iteration, if the simplified formula was satisfiable, then the original formula was satisfiable using the assignments so far (this could be formalized into an inductive argument). Once this was proven, we knew that when we exit the loop, the formula with all variables grounded simplifies to `true`, meaning that the original formula was satisfiable using the assignments we built.

- In many cases, you will only make one blackbox call to AlgB, in which case, you will want to prove an equivalence between the results of the blackbox call AlgB and the results of AlgA. To understand why you need to prove an equivalence statement, let's consider the reduction of 3-SAT (`ProbA`) to Independent Set (`ProbB`). In class, we developed a reduction to Independent Set by creating a specific graph G and returning "yes" exactly when Independent Set responded that "yes," there was an Independent Set of size $m + n$ on graph G. To prove the correctness of this reduction, we need to show that if there is an Independent Set of size $m + n$ on G (i.e. if we return "yes") then the formula from 3-SAT is satisfiable. We also need to show that if there is not an Independent Set of size $m + n$ on G then the formula is not satisfiable. Because it is usually easier to prove that something *does* exist (i.e. constructing an Independent Set or a satisfying assignment of variables) rather than proving that something *doesn't* exist, we will take the contrapositive of the second statement we need to prove. Now we need to prove the equivalence statement "there is an Independent Set of size $m+n$ on G if and only if the formula from 3-SAT is satisfiable." Both directions are proving that something exists, which means that we can prove both directions constructively. For example, in the first direction, we start with an Independent Set of size $m + n$ on G and show how we could construct an assignment of the variables from this Independent Set such that the 3-SAT formula is satisfiable.

Example: Baseball Elimination

Problem Statement Give an algorithm for Baseball Elimination that runs in $O(n^4)$ time by reducing to a Network Flow problem.
Input:

- A list of n baseball teams
- w_i for $i \in [n]$: the number of games that each team has won so far this season
- $r_{i,j}$ for $i < j; i, j \in [n]$: the number of games between teams i and j remaining in this season
- $T \in [n]$: the team we care about

Output:

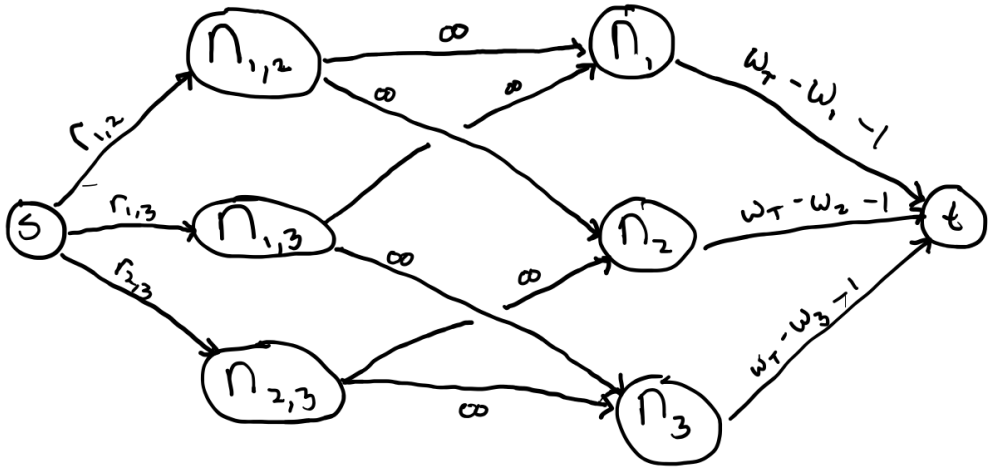
- "Yes" if it is possible for team T to have strictly more wins than all other teams by the end of the season.
- "No" if this is not possible.

Reduction Description Let's first note that our goal is to find out if it is possible for team T to have the most wins by the end of the season. So, we'll give team T the benefit of the doubt and assume that T wins all of its remaining games. This means we will increase w_T by the number of remaining games that T will play in, and we will set $r_{T,j}$ and $r_{i,T}$ to 0 for all $i, j \in [n]$.

We will now reduce this simplified Baseball Elimination problem to Integral Max Flow. We create our network N using the intuition that the flow represents remaining games and we pass one unit of flow to the winner of each game. First, we create a source s and a sink t. Next, to satisfy the constraint that each pair of teams i, j only has $r_{i,j}$ games remaining, we will create a "match-up" node $n_{i,j}$ for each pair of teams, and we will connect s to $n_{i,j}$ with an edge of capacity $r_{i,j}$. In order to limit how many games each team can still win (such that team T comes out with the most wins), we will create a node n_i for each team, and we will connect n_i to t with an edge capacity of $w_T - w_i - 1$. Since the flow through this edge corresponds to the number of remaining wins for team i, this means that team T must win more games than team i by the end of the season. We then create an edge between $n_{i,j}$ and n_i as well as an edge between $n_{i,j}$ and n_j, both with edge capacity ∞. In this way, flow represents wins getting assigned from each match-up to one of the two teams present in the match-up.

We use an Integral Max Flow blackbox to find the max-flow f on the network N. If $\nu(f) \geq \sum_{(i,j)} r_{i,j}$, we return "yes." Otherwise, we return "no." Note that we can use Integral Max Flow because we have integral capacities and we only care about integral flow (giving half of a win to one team and half to another is meaningless).

Figure 1: Example of the network N created for 3 baseball teams



Correctness For clarity, we'll refer to an assignment of wins and losses throughout the remaining games as a "result pattern," and we'll refer to the number of wins a team has by the end of the season as the total number of wins for that team.

For the first part of the correctness, we need to prove that the initial simplification of the problem in which we assign team T to win all of its remaining games is valid. We need to prove that there exists a result pattern in the original setup such that team T has a total number of wins larger than any other team $\iff$ there exists a result pattern in the simplified setup such that team T has a total number of wins larger than any other team. We leave this as an exercise.

Now we move on to the more important part of the correctness proof. We will work with our simplified setup in which team T has already won its remaining games. We will prove that there exists a result pattern such that team T has a total number of wins larger than any other team $\iff$ there is a max-flow on N of value $\nu(f) \geq \sum_{(i,j)} r_{i,j}$.

$\Rightarrow$ We are given a result pattern such that team T has a total number of wins larger than any other team, and we will construct a valid flow on network N of value $\sum_{(i,j)} r_{i,j}$. First, we send an amount of flow from s to $n_{i,j}$ equivalent to the number of games left to play between teams i and j. Since the edge capacity between s and $n_{i,j}$ is exactly $r_{i,j}$, we know that this flow is feasible so far. Then, let $z_{i,j}$ be the number of times that team i beats team j in the remaining games according to the result pattern we were given. We will send $z_{i,j}$ flow from $n_{i,j}$ to n_i. Since the edges from match-up nodes to team nodes have capacity ∞, this is feasible. Additionally, because exactly one of the two teams will win for each remaining game, we know $r_{i,j} = z_{i,j} + z_{j,i}$. Now, for each team node n_i we have $\sum_j z_{i,j}$ flow (corresponding to the number of wins for team i in all of the remaining games) that needs to be sent from n_i to t. Recall that the result pattern we started with gives team T a total number of wins larger than any other team. This means that $w_i + \sum_j z_{i,j} \leq w_T - 1$. Since $w_T - w_i - 1$ is the capacity of the edge from n_i to t, this means we can send all of the flow we needed to from node n_i to node t. In conclusion, since we sent $r_{i,j}$ flow from s to $n_{i,j}$ for each match-up node, this means that there exists a valid flow of value $\sum_{(i,j)} r_{i,j}$ through the network N.

$\Leftarrow$ We are given a max-flow on N of value $\nu(f) \geq \sum_{(i,j)} r_{i,j}$, and we will construct a result pattern such that team T has a total number of wins larger than any other team. Let $z_{i,j}$ be the flow on the edge between $n_{i,j}$ and n_i. We will assign team i to beat team j in $z_{i,j}$ remaining games. First, we will show that this is a valid result pattern. Note that because we use Integral Max Flow as our blackbox, we can assume that the flow will be integral, which is important since it would be invalid to assign only a portion of a win to one team. Additionally, because our flow is value $\nu(f) \geq \sum_{(i,j)} r_{i,j}$, and the total capacity of all outgoing edges from S is $\sum_{(i,j)} r_{i,j}$, we know that all edges from s to $n_{i,j}$ are used at full capacity $r_{i,j}$. Then, by the conservation of flow, we know that $z_{i,j} + z_{j,i} = r_{i,j}$ since there are only two outgoing edges from node $n_{i,j}$ (to node n_i and node n_j respectively). This means that our result pattern is valid, because for each pair of teams i and j, the number of times that i beats j plus the number of times that j beats i is exactly the number of remaining games. Next, we will show that this result pattern gives T a total number of wins larger than any other team. Note that by conservation of flow, each edge from n_i to t must have flow equal to $\sum_j z_{i,j}$. By the capacity constraints on n_i to t, we know $\sum_j z_{i,j} \leq w_T - w_i - 1$. Note that $\sum_j z_{i,j}$ is the total number of wins that we assigned i in the remaining games. Since we know $w_i + \sum_j z_{i,j} \leq w_T - 1$, this means that T has a total number of wins larger than any other team.

Running Time Analysis First, simplifying the problem such that we assign T to win all of its remaining games requires looping over $O(n)$ values of the type $r_{T,j}$ and $r_{i,T}$. The graph we build has $O(n^2)$ match-up nodes and $O(n)$ team nodes, for $O(n^2)$ nodes in total. The graph also has $O(n^2)$ edges because there are $O(n^2)$ match-up nodes, each of which has a constant number of edges incoming and outgoing, and $O(n)$ team nodes, each with one outgoing edge. The only calculations necessary to define edge capacities are calculating $w_T - w_i - 1$ for $O(n)$ teams. Finally, we only make one blackbox call, which returns a flow. We can calculate the value of this flow in $O(n^2)$ time by summing the flow over all edges leaving s. We then compare this to $\sum r_{i,j}$ (which also takes $O(n^2)$ time to evaluate). So, our entire reduction takes $O(n^2)$ time.

Further, our network is integral and we know that the Integral Max Flow problem can be solved by an algorithm taking $O(|V| \cdot |E|)$ time (where $|V|$ is the number of vertices in N and $|E|$ is the number of edges). This means that if we use this algorithm instead of the blackbox call in our reduction, calculating the flow of N will take $O(n^2 \cdot n^2)$ time. This yields an algorithm for Baseball Elimination that takes $O(n^4)$ time.