CS 3510 Algorithms 3/12/2024

Reduction

Lecturer: Jan van den Brand *Scribe(s): Ankith Thalanki*

Today was our first lecture on complexity theory. Reduction is a key part of complexity theory as by allowing us to transform problems into each other it allows us to compare their complexity. Reduction is the process of taking some problem and transforming it, generally in polynomial time, into another problem we have an algorithm for. The process is carried out in two steps: in the first we convert $\mathcal{I}_\mathcal{A}$ to $\mathcal{I}_\mathcal{B}$ with a polynomial time, then in the second we use our algorithm for problem B to solve the transformed input. We generally don't have to worry about converting the output, because in complexity theory we mostly focus on decision problems, which have "Yes" or "No" answers questions. In the below notes we mainly look at Shortest Product Path, and how it can be compared to the traditional Shortest Path problem via reduction.

Shortest Product Path

Let's say we have a graph with edge weights, G, which contains two connected points, s and t. The Shortest Product Path Problem asks the question, what is the path that minimizes the product of all the edges on the path? For example, a path going from s to e to t, through edges $e_1 = 4$, and $e_2 = 2$, would have a product cost of 8. To avoid additional complications, we are assuming all edges are positive, ensuring that paths don't become shorter by traversing additional edges. Using our knowledge of Dijkstra, an algorithm that can determine the shortest path sum in a graph with positive edges, we can probably create a new algorithm to solve our problem. But what if we didn't have access to Dijksta, or it way too complicated for us to alter? Would there be a way to solve our problem without altering Dijkstra or any other shortest path algorithm itself? We can do so through reducing our Shortest Product Path problem into a Shortest Sum Path problem.

Shortest Product Path to Shortest Sum Path

There are three steps necessary when converting the Shortest Product Path problem to the Shortest Sum Path problem, we first have to convert our input into a format suitable for the Shortest Sum Path. Then we have to call the function we are reducing to. Let us represent the Shortest Sum path in graph G between s and t as the value returned by ShortestPath(s, t, G). Finally, we need to convert our output from a sum into a product. As we want to minimize a product of edges via a function that minimizes sums, are there any mathematical functions that can easily convert a product into sum and vice-versa? Yes there is, and the key lies in some of the logarithms properties. Remember from pre-calculus that $\log(a)+\log(b) = \log(a*b)$. Notice that this indicates that minimizing the sum $\log(a)+\log(b)$ is equivalent to minimizing the product $\log(a*b)$. We can therefore transform the edge cost of each edge via the following transformation: $c'_e = \log(c_e)$. Thus, when ShortestPath(s, t, G) returns $min_{st-pathP} \sum_{e\in P}(c'_e) = min_{st-pathP} \sum_{e\in P}(log(c_e)) = min_{st-pathP} \log(\prod_{e\in P}(c_e))$,

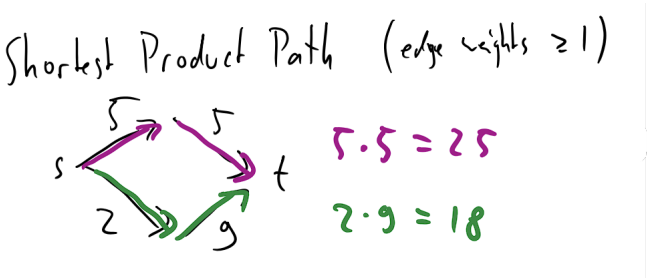

Figure 1: Example of the Shortest Product Path problem. While typically the purple path would be smaller, because $10 < 11$, because we are defining paths via the product of the edge weights, the green path would be shorter in this situation.

it is also returning the minimum product of the original weights. In order to recuperate the actual minimum product from the logarithm, we can simply just put the logarithm in the exponent of a power of 2 (assuming that all our logs have a base of 2). For example, $p = 2^{log_2(p)}$. The algorithm illustrating our reduction of the Shortest Product Problem into a Shortest Sum Problem is below.

Algorithm 1 Shortest Product Path

Input: G, s, t

for $e \in G$ **do**
  $c'_e = \log(c_e)$
end
$distance = \text{ShortestPath}(s, t, G)$
$distance' = 2^{distance}$
return $distance'$

Decision vs. Optimization Problems

An **optimization problem** is one where the answer is the maximum or minimum of some quantity, like "What is the shortest path in graph G?" or "What is the maximum element of this array?". A **decision problem** on the other hand is a "yes" or "no" problem, like "Is there an element smaller than 3 in this array?" or "Is the shortest path between these two nodes smaller than 4?". While decision problems may seem much easier than optimization problems, and they are to a certain extent, for the purpose of complexity theory, the difference is negligible. This is because many optimization problems can be reduced to the decision version of the problem in polynomial time and vice versa, and in complexity theory we don't compare about minute details, mostly about whether a problem can be solved in polynomial time, exponential time, or worse. For example, let us define the decision version of the ShortestProductPath as asking the question, "Is the shortest product path in G between s and t shorter than d?". Then our solution to the problem would simply be $d <$ ShortestProductPath(s, t, G). Notice that because decision problems have such a simple output, converting the output from one decision problem to another in reduction is

negligible, and therefore we only have to worry about the complexity of transforming the input when we are reducing a decision problem to another decision problem. This helps complexity theorists worry about the complexity of a problem instead of the complexity of its output, which is why most of complexity theory is focused on decision problems.