AM 221: Advanced Optimization Spring 2016

Prof. Yaron Singer *Section 9 — Wednesday, Mar. 30th*

As we move to the last part of the course dedicated to combinatorial optimization, it is important to understand how to reason about the complexity of an algorithm. Considerations about the efficiency of computation were already implicitly present when we discussed Linear Programming and Convex Optimization. The goal of today's section is to make these discussions more formal by giving a short introduction to the theory of computational complexity.

1 Efficiency of Computation

1.1 Time Complexity

Let us consider the pseudo-code of the greedy algorithm seen in class to solve the KNAPSACK problem.

Algorithm 1 Greedy algorithm for KNAPSACK

```
Require: item values $v_1, \dots, v_n$, item costs $c_1, \dots, c_n$, budget $B$
 1: sort items by decreasing value of $\frac{v_i}{c_i}$
 2: $S \leftarrow \emptyset$, $c \leftarrow 0$, $i \leftarrow 1$, $v \leftarrow 0$
 3: while $c \leq B$ do
 4:     $S \leftarrow S \cup \left\{\frac{v_i}{c_i}\right\}$
 5:     $i \leftarrow i + 1$
 6:     $v \leftarrow v + v_i$
 7: end while
 8: $j \leftarrow \text{argmax}_{1 \leq i \leq n} v_i$
 9: if $v_j > v$ then
10:     return $\{j\}$
11: else
12:     return $S$
13: end if
```

The running time of this algorithm is measured in terms of the number of machine operations it performs on an input of size n, in the worst case over all possible inputs. For the algorithm above, the sorting on line 1 can be done in $c_1 \cdot n \log n$ operations for some constant c_1. In the worst case, the while loop is repeated n times, each repetition of the while loop requires a small number c_2 of operations. Computing the maximum in line 8 takes $c_3 \cdot n$ operations (we need to visit each element i, $1 \leq i \leq n$). Finally, there is a constant number c_4 of operations for the initializing at the beginning and the IF test at the end to decide which answer to return. Overall the number of operations is $T(n) = c_1 \cdot n \log n + c_2 \cdot n + c_3 \cdot n + c_4$.

The exact constants appearing in $T(n)$ is not relevant: these could vary depending on the exact details of the implementation, the type of processor executing the algorithm, etc. For this reason, time complexity is measured as the asymptotic order of magnitude as n grows to infinity. This is exactly capture by the following definition of O (big O):

Definition 1. *For a function $g : \mathbb{N} \to \mathbb{R}$, we define:*

$$O(g) \overset{\text{def}}{=} \{f : \mathbb{N} \to \mathbb{R} \mid \exists c, n_0 \ s.t. \ |f(n)| \leq c \cdot |g(n)| \ \forall n \geq n_0\}$$

By slightly abusing notations, we usually right $f(n) = O\big(g(n)\big)$ to mean $f \in O(g)$.

Definition 2. *An algorithm runs in* ***polynomial time*** *if there exists some constant $c \geq 1$ s.t. the running time of the algorithm is $O(n^c)$, where n is the size of the input.*

Example. For the greedy algorithm presented above, the term $n \log n$ dominates the other terms, hence we have $T(n) = O(n \log n)$. Since $\log n \leq n$ for all positive integers, we also have $T(n) = O(n^2)$, *i.e* this algorithm runs in polynomial time.

Remark. What do we mean by *size of the input*? This is the number of bits required to represent the input. For example, for the Knapsack problem, the value v_i of item i can be represented by using $\log v_i$ bits. So the overall size of the input is $\log B + \sum_{i=1}^{n}(\log v_i + \log c_i)$.

Another example is when the input is a graph with n nodes and m edges. Each node can labeled using $\log n$ bits. We can describe each edge by writing the labels of its endpoints, so the overall number of bits is $m \log n$.

In this class, we will usually consider an algorithm to be efficient if it runs in polynomial time (of course, the smaller the exponent of the polynomial the better).

1.2 Approximation algorithms

Let us now revisit some of the results we saw on the complexity of certain optimization problems. First, we need to the following definition of an approximation algorithm.

Definition 3. *Let us consider the following maximization problem: $\max_{x \in K} f(x)$ for some value function $f : K \to \mathbb{R}$. An algorithm is said to be a α-**multiplicative approximation algorithm** if it returns $y \in K$ such that:*

$$f(y) \geq \alpha \max_{x \in K} f(x)$$

*An algorithm is said to be an ε-**additive approximation algorithm** if it returns $y \in K$ such that:*

$$\max_{x \in K} f(x) \leq f(y) + \varepsilon$$

In other words, the approximation of the algorithm can be measured either as an absolute error or a relative error. Note that when we have bounds on the range of values taken by f, it is possible to convert one to the other.

The simplex algorithm that we discussed in class for Linear Programming does not run in polynomial time: in the worst case, it might visit exponentially vertices of the feasible set. It was believed for a long time than Linear Programs could not be solved in polynomial time. The breakthrough result from Leonid Khachiyan showed that the *ellipsoid algorithm* solves linear programs in polynomial time.

Theorem 4 (Khachiyan, 1979). *For any linear program over n variables, the ellipsoid algorithm finds an exact solution (or a proof of infeasibility/unboundedness) in $O(n^6 L)$ arithmetic operations on $O(L)$ digit numbers. Using FFT-based multiplication, each arithmetic operation can be effectuated in time $O(L \cdot \log L \cdot \log\log L)$, so the overall running time of the ellipsoid algorithm is $O(n^6 L \cdot \log L \cdot \log\log L)$.*

The exponent of n has been further improved in subsequent research. For example, the famous Karmarkar's algorithm reduced it to $n^{3.5}$.

For convex programs, the analysis of the gradient descent algorithm for strongly convex functions proves the following theorem.

Theorem 5. *Consider the problem $\min_{\mathbf{x}\in\mathbb{R}^n} f(\mathbf{x})$ where f is a strongly convex function. Then, for any $\varepsilon > 0$, the gradient descent algorithm is an ε-additive approximation algorithm and runs in time $O(p(n)\log\frac{1}{\varepsilon})$ where p is polynomial in n.*

The polynomial p hides the strong convexity constants of f as well as the time required to compute f and its gradient on any given input. Interestingly, we see that we can obtain an ε-approximation for any $\varepsilon > 0$, the cost being that the running time depends on $\frac{1}{\varepsilon}$. In other words, we can think of the algorithm as being parametrized by ε: this is called an **approximation scheme**. The quality of an approximation scheme is then measured by the dependency of the complexity as a function of $\frac{1}{\varepsilon}$. In convex optimization, $\log\frac{1}{\varepsilon}$ is referred to as *linear convergence*. In contrast, the analysis of gradient descent we did in Problem Set 6 for functions with a Lipschitz-continuous gradient only gave a running time complexity of $O(p(n)\frac{1}{\varepsilon})$.

We saw another example of an approximation scheme in class for the KNAPSACK problem: for any $\varepsilon > 0$, there exists a $(1-\varepsilon)$-multiplicative approximation algorithm for the Knapsack problem which runs in time $O(\frac{n^3}{\varepsilon})$ where n is the number of items.

2 A Glimpse into the Beautiful Theory of Computation

2.1 Decision problems, Optimization Problems

The theory of computational complexity usually considers *decision problems* as defined below.

Definition 6. *A **decision problem** is a function $Q : \{0,1\}^* \to \{YES, NO\}$. In other words, the input of a decision problem is a sequence x of bits, and the output is either YES or NO.*

The problems we discussed above were optimization problems. Any optimization problem can be reformulated as a decision problem; for example let x be the bit representation of an instance of the KNAPSACK problem, then we can formulate the following decision problem for any $V \in \mathbb{R}$: $Q(x) =$*does there exist a subset of the n items of cost at most B and value at least V?*

It is clear that if we know how to solve an optimization problem exactly, solving the associated decision problem can be done in the same amount of time: for KNAPACK simply compute the optimal set of items and compare its value to V.

Conversely, an algorithm to solve a decision problem can usually be used to solve the associated optimization problem. Without giving a formal statement, let us illustrate this with the KNAPSACK problem:

- first note that the optimal value of KNAPSACK is upper bounded by $A = \sum_{i=1}^{n} v_i$. The optimal value can thus be found by binary search over the range $[0, A]$ by solving $\log A$ instances of the decision problem.
- once the optimal value V is known, an optimal set of items can be found in the following way: for each item i, solve the decision problem with value V after removing item i. If the answer if NO, then we know that i belongs to the optimal set of items.

Furthermore, note that the above way to solve the optimization version of KNAPSACK only requires solving $n \log A$ instances of the decision problem. Hence, if the decision problem can be solve in polynomial time, then the optimization problem can also be solved in polynomial time.

2.2 The complexity classes P and NP

Definition 7. *The complexity class **P** is the set of all decisions problems solvable through a polynomial time decision problem. In other words, for an input x the algorithm runs in time $O(|x|^c)$ for some $c \geq 1$.*

As we will discuss below, it is sometimes hard (or impossible) to find a polynomial time algorithm for some problems. A weaker notion is the one of a *verifier.*

Definition 8. *A verifier for a decision problem Q is an algorithm A which takes two input x and c where $|c| = O(|x|^a)$ for some $a \geq 1$ and is such that:*

- *if $Q(x) =$ YES, then there exists c such that $A(x, c) =$ YES*
- *if $Q(x) =$ NO, then for all c, $A(x, c) =$ NO.*

In other words, c can be thought of a *certificate* or a *proof* that the algorithm A can use to verify that the input x is a YES instance. The proof needs to be correct: if the instance is a NO instance, then no proof can convince the algorithm that it is a YES instance.

Definition 9. *The complexity class **NP** is the class of all decision problems which admit a polynomial-time verifier.*

Example. As we will discuss below, there is no known algorithm to solve the Knapsack problem in polynomial time, so we don't know whether or not KNAPSACK is in P. However it is easy to see that KNAPSACK$\in$ NP. For an input x for which the answer is YES (there exists a set of items whose value if at least V and whose cost is at most B), a possible certificate is simply a set S of items of cost at most B and of value V. Then the verifier A proceeds as follows:

- compute $v(S)$ the sum of the values of the items in S and $c(S)$ the sum of the costs of the items in S.

- if $c(S) \leq B$ and $v(S) \geq V$ output YES, otherwise output NO.

It is clear that if x is a NO instance, then no certificate can convince the verifier that x is a YES instance.

It is intuitively clear that a verifier for a given problem is not as powerful as an algorithm which solves this problem. In fact, we have the following proposition.

Proposition 10. *$P \subseteq NP$.*

Proof. Let Q be a decision problem in P and let A be an algorithm which solves Q in polynomial time. Then we construct the following polynomial time verifier B for Q: given input (x, c), $B(x, c) = A(x)$. In other words, the verifier simply ignores the certificate c and outputs the answer given by A on this input. It is easy to verify that B is a polynomial time verifier for Q which shows that $Q \in$ NP. □

2.3 Reductions

A fundamental concept in computability is that of reducibility: it is a way to compare two problems and show that one is harder or simpler than the other.

Definition 11. *Let Q and R be two decision problems. We say that Q is poly-time reducible to R and we write $Q \leq_P R$ iff there exists a polynomial-time algorithm A such that for any input x to problem Q, we have $Q(x) = R\big(A(x)\big)$.*

The way to think about this definition is the following: the algorithm A transforms an instance of problem Q into an instance of problem R such that the output of R on this transformed instance is the same as the output of Q on the original instance. In other words, once we have an algorithm to solve R and the reduction algorithm A, then we can solve problem Q. We interpret this as saying that Q is easier to solve than R, because an algorithm which solves R can be used to solve Q.

Let us now give an example of a reduction. Let us consider the two following problems:

KNAPSACK

- **input:** values $v_1, \dots, v_n$, costs $c_1, \dots, c_n$, budget B, target value V
- **question:** does there exist a subset S of items such that $\sum_{i \in S} v_i \geq V$ and $\sum_{i \in S} c_i \leq B$.

PARTITION

- **input:** values $v_1, \dots, v_n$
- **question:** does there exist a partition (S, T), with $S \cap T = \emptyset$ and $S \cup T = \{1, \dots, n\}$ such that $v(S) \overset{\text{def}}{=} \sum_{i \in S} v_i = \sum_{i \in T} v_i \overset{\text{def}}{=} v(T)$.

Proposition 12. PARTITION$\leq_P$KNAPSACK

Proof. Let us first define the reduction algorithm A: given an input to Partition we construct the following input to Knapsack: the set of items and their values stay the same. We define the cost c_i of item i by $c_i = v_i$. Finally we define $V = \frac{1}{2}\sum_{i=1}^n v_i$ and $B = \frac{1}{2}\sum_{i=1}^n v_i$. It is clear that this reduction can be computed in polynomial time.

Let us now prove that an input x to Partition has answer YES iff $A(x)$ has answer YES to Knapsack.

Let us consider a YES instance of Partition and let us denote by (S, T) the partition of the items such that $v(S) = v(T)$. Then we have $v(S) = v(T) = \frac{1}{2}\sum_{i=1}^n v_i \geq V$ and $c(S) = c(T) = \frac{1}{2}\sum_{i=1}^n c_i \leq B$. Hence the corresponding Knapsack instance is a YES instance.

Let us now consider a YES instance of Knapsack, and consider a feasible solution S, that is, a set S such that $v(S) \geq V$ and $c(S) \leq B$. But then, because $c_i = v_i$ and $B = V$, this implies $\sum_{i\in S} v_i = \frac{1}{2}\sum_{i=1}^n v_i$. Hence $(S, \bar{S})$ (where $\bar{S}$ represents the complement of S) is a partition of the items such that each part has the same value. We have just shown that the corresponding Partition instance is a YES instance. □

2.4 NP-completeness

Definition 13. *A problem Q is **NP-Complete** iff:*

1. *Q is in NP, and*
2. *$\forall R \in NP$, $R \leq_P Q$*

In other words, NP-complete problems can be thought of as the hardest problems in NP, because every other problem in NP can be reduced to them. The most important open question in the theory of computation is whether the two classes P and NP are equal to each other, *i.e* is the inclusion of Proposition 10 strict or not? Despite knowing many NP-complete problems, no polynomial time algorithm to solve any of them has been found until now, which is why it is widely believed that the inclusion is strict. If this is the case, then NP-complete problems cannot be solved in polynomial time. Until we finally get the answer to this question, the best we can say is that NP-complete problems are not known to be solvable in polynomial time.

How to show that a problem is NP-complete? The following proposition gives a powerful for this.

Proposition 14. *Let Q and R be two decision problems in NP. If Q is NP-complete and $Q \leq_P R$ then R is NP-complete.*

Proof. We simply need to show that for any problem $T \in$ NP, $T \leq_P R$. Let A be a reduction algorithm from T to Q (exists because Q is NP-complete) and let B be a reduction algorithm from Q to R (exists by assumption). Then it is easy to see that the algorithm which maps x to $B(A(x))$ (*i.e* running A on input x and then B on the output of A) is a reduction from T to R. □

What the proof just showed is that reductions can be composed, which implies that the $\leq_P$ sign is transitive. A consequence of Proposition 14 is that to show that a problem R is NP-complete, we simply need to show that is in NP and exhibit another NP-complete problem which can be reduced to it. However, we are still facing a chicken-and-egg problem: we need to know at least one NP-complete problem to begin with.

The famous Cook's theorem proved by Stephen Cook in 1971 proved that the SAT problem is NP-complete. The following year Richard Karp gave a list of 21 NP-complete problems obtained by reduction (or chains of reductions) from the SAT problem. Both Richard Karp and Stephen Cook obtained the Turing Award for their works on the theory of computation.

The implication for us is that we can prove the NP-completeness of a problem by reducing a well-known NP-complete problem to it. For example the PARTITION problem introduced above is known to be NP-complete (it is in fact one of Karp's 21 NP-complete problems). Combining this fact with Proposition 12, we obtain:

Proposition 15. KNAPSACK *is NP-complete.*