6.854 Advanced Algorithms

Lecture 15: 10/29/2004 Lecturer: David Karger

Scribes: John Provine

In this lecture we continue the discussion of treewidth and begin our study of online algorithms.

Treewidth

15.1 Review

The treewidth of a graph measures roughly "how close" a graph is to a tree. The definition of treewidth follows from the definition of a tree decomposition.

Definition 1 A *tree decomposition* of a graph $G(V, E)$ is a pair (T, χ) where $T = (I, F)$ is a tree and $\chi = \{\chi_i | i \in I\}$ is a family of subsets of V such that (a) $\bigcup_{i \in I} \chi_i = V$, (b) for each edge $e = \{u, v\} \in E$, there exists $i \in I$ such that both u and v belong to χ_i, and (c) for all $v \in V$, there is a set of nodes $\{i \in I | v \in \chi_i\}$ that forms a connected subtree of T. The *width* of a tree decomposition (T, χ) is given by $\max_{i \in I}(|\chi_i| - 1)$.

The *treewidth* of a graph G equals the minimum width over all possible tree decompositions of G.

Treewidth is often characterized in terms of elimination orderings. An *elimination ordering* of a graph is an simply an ordering of its vertices. Given an elimination ordering of a graph G, one can construct a new graph by iteratively removing the vertices v in order and adding an edge between each of v's neighbors. The induced treewidth of this elimination ordering is the maximum neighborhood size in this elimination process. The treewidth of G is then the minimum induced treewidth over all possible elimination orderings. For example, the treewidth of a tree is 1, and the treewidth of a cycle is 2 (each time you remove a vertex, you connect its two neighbors to form a smaller cycle).

In general, the problem of computing the treewidth of a graph is NP-complete. However, the problem is fixed-parameter tractable with respect to treewidth itself.

15.2 Fixed-parameter tractability of SAT

In this section, we will show that the problem of *satisfiability* (SAT) is fixed-parameter tractable with respect to treewidth. Clearly, we need to define the graph that contains our treewidth parameter.

Definition 2 Given a CNF formula F (i.e. a conjunction of clauses), the *primal graph* of F, $G(F)$, is the graph whose vertices are the variables of F, where two variables are joined by an edge if they appear in the same clause.

Suppose that the primal graph $G(F)$ has treewidth k. The proof below provides an algorithm that, given an optimal elimination ordering $x_1, x_2, \ldots, x_n$ of the vertices of $G(F)$, solves SAT in time $O(nk \cdot 2^k)$. Though we have restricted our attention to SAT, a similar approach can be used for any optimization problem that has an associated graph with local dependency structure.

Claim 1 *SAT is fixed-parameter tractable with respect to treewidth.*

Proof: Let F be a CNF formula and G its primal graph. As in the tree decomposition procedure, eliminate the variables $x_1, x_2, \ldots, x_n$ in that order to construct a new graph G^*. When we eliminate x_i, we remove all edges incident to x_i in G, and add any non-existing edges between a pair of neighbors of x_i in G^*. Note that we can form a new CNF formula in which we combine all clauses containing x_i into one by taking the disjunction of all the literals in those clauses, except those containing the variable x_i. It is clear that G^* is the primal graph for the new formula.

For each new clause in this procedure, construct a truth table for that clause. The truth table is a list of all satisfying assignments of variables in the clause which is a partial satisfying assignment for all the original clauses. For example, if $(x \vee y \vee z)$ and $(\neg x \vee \neg y \vee z)$ are the only two clauses containing x, then on eliminating x, we form the new clause $(y \vee z \vee \neg y)$. The truth table for the new clause is derived from truth tables for the original clauses as follows:

$(\ x$	$\vee\ y$	$\vee\ z\ )$	$(\ \neg x$	$\vee\ \neg y$	$\vee\ z\ )$
1	1	1	1	1	1
0	1	1	0	1	1
1	0	1	1	0	1
0	0	1	0	0	1
1	0	0	1	0	0
0	1	0	0	1	0
1	1	0	0	0	0

$$\longrightarrow$$

$(\ y$	$\vee\ z$	$\vee\ \neg y\ )$
1	1	–
0	1	–
1	0	–
0	0	–

Repeating this procedure for each variable eliminated, we will eventually end up with a single truth table representing the entire formula. If the table has at least one entry, then there is an assignment which satisfies all the original clauses. In this case, we return *true*. Otherwise, F is not satisfiable, so we return *false*.

Since G has treewidth k, there are at most k neighbors of any vertex in an optimal elimination ordering, so we construct a truth table with at most $k \cdot 2^k$ entries per vertex eliminated. Since there are n vertices, this yields a total running time of $O(nk \cdot 2^k)$. ■

Online Algorithms

Parts of this section are adapted from Kevin Zatloukal's scribe notes from 2003.

15.3 Introduction

All of the algorithms we have studied so far in this course have taken their entire input up front and used it to compute an answer. In this section, we will study algorithms that are given the input one piece at a time and are forced to make a decision solely based on the input they have read so far. The goal is for the quality of those decisions to be close to the quality we would achieve if we were given the entire input up front. We call algorithms that get the input one piece at a time *online* algorithms and those that get the input all at once *offline* algorithms.

15.3.1 Competitive analysis

In an online problem the input is given as a sequence $\sigma = \sigma_1\sigma_2\cdots\sigma_k$ of *requests* or *events.* Each request σ_i must be processed immediately and incurs some cost. Let us denote the total cost of an algorithm A on input σ by $C_A(\sigma)$ and the optimal cost for this particular input by $C_{\text{MIN}}(\sigma)$. As mentioned above, our focus will be on finding online algorithms for which the total cost can be guaranteed to be relatively close to the optimal cost. The following definition makes this notion precise.

Definition 3 An (deterministic) online algorithm A has *competitive ratio k* (or A is *k-competitive*) if for all inputs σ,

$$C_A(\sigma) \leq kC_{\text{MIN}}(\sigma) + O(1).$$

The smallest such k is often referred to as the *regret ratio.*

This defines the competitive ratio in a worst-case sense, as is usual for deterministic algorithms. For randomized algorithms, we will consider the expected cost of the algorithm.

Definition 4 An randomized online algorithm A has *competitive ratio k* if for all inputs σ,

$$\mathrm{E}[C_A(\sigma)] \leq \alpha C_{\text{MIN}}(\sigma).$$

The analysis of online algorithms is similar to that of approximation algorithms in that we are comparing the quality of our solution to that of the (possibly unknown) optimal algorithm. As with approximation algorithms, our focus is on the quality of the solution rather than on running time or space (beyond basic polynomiality). Furthermore, in both cases, we are comparing our algorithm against an optimal algorithm that can cheat. In approximation algorithms, the optimal algorithm may not run in polynomial time. In online algorithms, the optimal algorithm can see the future because it is given all of the input up front.

In todays lecture, we examine two examples of online algorithms: the *ski rental problem* and *(online) load balancing.*

15.4 The Ski Rental Problem

After finals week, suppose that you head to a ski resort. You have the entire vacation as well as the Independent Activities Period to ski. Unfortunately, you know from past experience that, at some point, the fun will come to a premature end when fate steps in and breaks your leg. On each day until then, you have to make an important decision: should you rent ski equipment for 1 dollar or buy your own for T dollars? If you keep renting long enough, you will eventually find that you have spent more than T dollars, so it would have been cheaper to buy your own equipment at the beginning. However, if you buy your own, then you might break your leg that very day, wasting $T - 1$ dollars.

One idea would be to always buy on the first day. However, if you break your leg that day, then you spent T dollars while the optimum algorithm would have rented and spent only 1 dollar, so this algorithm is only T-competitive. A better idea is to rent for T days and then buy on day $T + 1$. To analyze this algorithm, suppose that you break your leg on day d. If $d \leq T$, then we always rented, which was the optimal decision. If $d > T$, then we will pay $2T$. The optimal decision would have been to buy on the first day, which would cost T dollars. But we only spent twice that, so this algorithm is 2-competitive.

15.5 Online Load Balancing

In the online load balancing problem, the goal is to assign a sequence of jobs to m identical machines. The jobs arrive at different times, and we must immediately choose the machine to which we will assign it. Each job comes with a load, the amount of computational resources it will consume. And all jobs are permanent—they never terminate. As in the offline problem, the goal is to minimize the maximum load on any machine.

Although we did not present the algorithm in this way originally, we have already seen one example of an online algorithm for load balancing: *Graham's algorithm.* In this algorithm, when each job arrives, assign it to the least loaded machine. Since this algorithm does not require looking at any of the jobs yet to arrive, it is an online algorithm. We proved before that, over any sequence of jobs, the maximum load produced by our algorithm is at most twice the maximum load of the optimal solution. This means that the algorithm is 2-competitive.

In fact, the competitve ratio is exactly $2 - \frac{1}{m}$, where m is the number of machines. The fact that greedy is not better than $2 - \frac{1}{m}$ is shown by a simple example which is $m(m - 1)$ unit size tasks followed by one task of size m.

15.5.1 Improvements to Graham's algorithm

There has been a surprisingly large amount of research towards achieving a better competitive ratio for the online load balancing problem. In 1992, Fiat et al. found an algorithm with competitive ratio $2 - \epsilon$ for a small constant $\epsilon \approx \frac{1}{70}$. In 1994, Karger, Phillips and Torng used an LP solver to achieve a competitive ratio of 1.945. In 1997, Albers found a 1.923-competitive algorithm.

It is worth noting that one may consider the case where the value of the optimum cost is known. In this case, the lower bound is provably 4/3 (the competitive ratio is exactly 4/3 for two machines). The best known deterministic algorithm for this instance, due to Azar and Regev, is 1.625-competitive.

See [**?**] for an overview of online load balancing.