CS880: Approximations Algorithms

Scribe: Siddharth Barman **Lecturer:** Shuchi Chawla
Topic: Edge Disjoint Paths; Dynamic Programming **Date:** 02/01/07

In this lecture we give an algorithm for Edge disjoint paths problem and then discuss dynamic programming.

4.1 Edge disjoint paths

Problem Statement: Given a **directed graph** G and a set of terminal pairs $\{(s_1, t_1), (s_2, t_2), \cdots, (s_k, t_k)\}$, our goal is to connect as many pairs as possible using non edge intersecting paths.

Edge disjoint paths problem is $\mathcal{NP}$-Complete and is closely related to the multicommodity flow problem. In fact integer multicommodity flow is a generalization of this problem. We describe a greedy approximation algorithm for the edge disjoint path problem due to Jon Kleinberg [4].

Algorithm: Compute shortest path distance between every (s_i, t_i) pair. Route the one with smallest distance along the corresponding shortest path, remove all the used edges from the graph and repeat.

Theorem 4.1.1 *The above algorithm achieves an $O(\sqrt{m})$ approximation, where m is the number of edges in the given graph.*

Before we dwell on the proof of the above theorem we present an instance of the problem (Figure 1) for which the greedy algorithm gives an $\Omega(\sqrt{m})$ approximation. This shows that the analysis is in fact tight.

See Figure 1; The graph is constructed such that the length of the path between terminal vertices s_{l+1}, t_{l+1} is smaller than all other (s_i, t_i) paths. Hence the greedy algorithm picks the path connecting s_{l+1} and t_{l+1} at the first go. This in turn disconnects all other terminal pairs. Thus the greedy algorithm returns a single path, but we can connect (s_i, t_i) pairs for all i between 1 and l by edge disjoint paths.

Note that for the construction to go through length of the path between s_{l+1} and t_{l+1} must be at least l and so the length of the shortest path between s_i and t_i for $1 \leq i \leq l$ must be more than l. So $m = O(l^2)$ and the approximation achieved is $l = \Omega(\sqrt{m})$.

Relation between the optimal solution and our greedy algorithm is achieved by charging each path in OPT to the *first* path in ALG that intersects it. For this we define short and long paths. A *short path* is one which has no more than k edges. Rest of the paths shall be referred to as *long paths*. We will pick an approximate value of k later.

Lemma 4.1.2 *OPT has no more than m/k long paths, where m is the number of edges.*

Proof: The paths in OPT are edge disjoint hence m/k paths of length more than k will cover all the m edges. ■

Lemma 4.1.3 *Each short path in OPT gets charged to some short path in ALG.*

Proof: The greedy algorithm picks the shortest path which is still available. Say P_G is the path

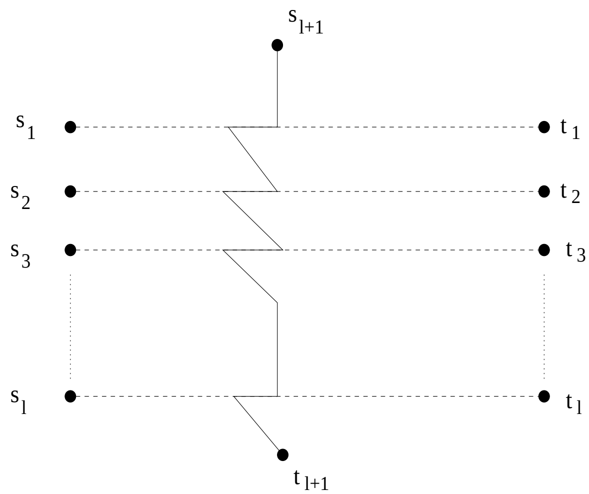

Figure 4.1.1: Bad example for greedy algorithm

picked up ALG that "cuts" P_{OPT} a fixed short path in OPT for the first time. As described earlier P_{OPT} gets charged to P_G. At that point of time all of previously selected paths of ALG are edge disjoint with P_{OPT}, hence P_{OPT} is still available, but ALG decides to choose P_G which implies that length of P_G is less than k, i.e. it is a short path. ■

Lemma 4.1.4 *Each short path in ALG gets charged at most k times.*

Proof: Paths of OPT are edge disjoint themselves hence in the worst case each edge of a short path selected by ALG cuts a different path of OPT. This bound the charge to k. ■

Next we use the above mentioned lemmas to prove Theorem 4.1.1.

Proof of Theorem 4.1.1**:** We partition the optimal solution in long and short paths i.e. $OPT = OPT_{long} + OPT_{short}$. By lemma 4.1.2 we have that OPT_{long} is at most m/k and using the other two lemmas we can bound OPT_{short} by $ALG \times k$. Hence,

$$OPT \leq \frac{m}{k} + ALG \times k$$

Setting $k = \sqrt{m}$ and noting that $ALG \geq 1$ we get

$$OPT \leq 2\sqrt{m} \times ALG$$

Hence the algorithm achieves an approximation factor of $2\sqrt{m}$. ■

The above algorithm and analysis is from Kleinberg's thesis [4]. Chekuri and Khanna [2] gave a better analysis of the same algorithm in terms of the number of vertices of the graph. In particular they showed that the greedy algorithm achieves an approximation of $O(n^{4/5})$ in general and an $O(n^{2/3})$ approximation for undirected graphs.

Surprisingly with complexity theoretic assumptions the greedy algorithm turns out to be the best one could hope for, although the same factor can also be achieved using linear programming.

Hardness results for the edge disjoint paths problem show that unless $\mathcal{P} = \mathcal{NP}$, in directed graphs it is not possible to approximate the edge disjoint path problem better than $\Omega(m^{1/2-\epsilon})$ for any fixed $\epsilon \geq 0$ [3]. For undirected graphs, edge disjoint paths cannot be approximated better than $\Omega(\log^{1/3-\epsilon} m)$ exists unless $\mathcal{NP} \subseteq \mathcal{ZTIME}(n^{\text{polylog}\, n})$ [1].

4.2 Dynamic Programming: Knapsack

The idea behind dynamic programming is to break up the problem into several subproblems, solve these optimally and then combine the solutions to get an optimal solution use them to solve the prob at hand. Generally for approximation we do not use dynamic programming to solve the given instance directly. We use two approaches. First we morph it into an instance with some special property and then apply dynamic programming to solve the special instance exactly. The approximation factor comes from this morphing.

Secondly, dynamic programming can as well be viewed as a clever enumeration technique to search through the entire solution space. With this in mind, approximation algorithms can be designed that restrict the search to only a part of the solution space and not the entire space and apply dynamic programming over this subspace. In this case the approximation factor reflects the gap between the overall optimal solution and the optimal solution over the subspace. Over the next two lectures we will see both kinds of techniques used.

We proceed to design an approximation algorithm for the knapsack problem which uses the morphing idea. Note that knapsack is known to be $\mathcal{NP}$-complete.

Problem Statement: Given a set of n items each with a weight w_i and profit p_i, along with a knapsack of size B our goal is to find a subset of items of total weight less than B and maximum total profit.

Knapsack can be solved exactly using dynamic programming. The exact algorithm proceeds by filling up a $n \times B$ matrix recursively. Each entry (i, b) in the matrix corresponds to the maximal profit that can be achieved using elements 1 through i with total weight less than b. Each entry takes a constant amount of time hence the time complexity is $O(nB)$. We can also employ another exact algorithm. This time we fill up an $n \times P$ matrix M, where $P = \sum_i p_i$ is the total profit. An entry (i, p) of M holds the value of the minimum possible weight required to achieve a profit of p using elements 1 through i. This algorithm takes time $O(nP)$.

These exact algorithms fall in the class of *pseudo polynomial time* algorithms. Formally, a pseudo polynomial time algorithm is one that takes time polynomial in the size of the problem in *unary*. Problems that have pseudo-poly time algorithms and are $\mathcal{NP}$-Hard are called *weakly $\mathcal{NP}$-Hard*.

We now describe how to obtain a polytime approximation algorithm. The main idea is to modify the instance so as to reduce $P = \sum_i p_i$ to some value that is bounded by a polynomial in n. In particular, we pick $K = n/\epsilon$ to be the new maximum profit, for some $\epsilon > 0$. We *scale* the profits uniformly, such that the max. profit equals K., and then we round down these scaled values to the nearest integer to ensure that we have integer profits i.e.

$$p_i' = \left\lfloor p_i \times \frac{K}{p_{max}} \right\rfloor$$

We then solve the knapsack exactly on the new profit values p'_i. We now show that we do not loose much in rounding.

Theorem 4.2.1 *The above mentioned algorithm achieves a* $1+\epsilon$ *approximation.*

Proof: Let O denote the value of the optimal solution OPT on the original instance and O' denote the value of OPT on new instances, i.e. $O = \sum_{i \in OPT} p_i$ and $O' = \sum_{i \in OPT} p'_i$.

Similarly A and A' be the value of the solution obtained by the algorithm on original and new instances respectively.

Note that $p_i \geq \frac{p_{max}}{K} \left\lfloor p_i \times \frac{K}{p_{max}} \right\rfloor$. Hence

$$\sum_{i \in ALG} p_i \geq \sum_{i \in ALG} \frac{p_{max}}{K} \left\lfloor p_i \times \frac{K}{p_{max}} \right\rfloor$$

Which is equivalent to $A \geq \frac{p_{max}}{K} \times A'$. We solve the problem exactly on the new instances hence A' is the optimal value for p'_is. Hence $A' \geq O'$. Combining the two inequalities we get $A \geq \frac{p_{max}}{K} O'$. Expanding O' we get the following

$$\begin{aligned} \frac{p_{max}}{K} O' &= \frac{p_{max}}{K} \sum_{i \in OPT} \left\lfloor p_i \times \frac{K}{p_{max}} \right\rfloor \\ &\geq \frac{p_{max}}{K} \sum_{i \in OPT} \left(\frac{p_i K}{p_{max}} - 1 \right) \\ &= \sum_{i \in OPT} p_i - n \frac{p_{max}}{K} \\ &= O - \epsilon p_{max} \\ &\geq O(1-\epsilon) \end{aligned}$$

Hence $A \geq O(1-\epsilon)$

■

Note that the above mentioned algorithm takes $O(p_{max} n^2)$ time, hence the algorithm runs in time $poly(n, \frac{1}{\epsilon})$. Such algorithms belong to the so called $FPTAS$ (Fully Polynomial Time Approx. Scheme) class. In general we have the following two relevant notions:

Definition 4.2.2 *FPTAS (Fully Polynomial Time Approx. Scheme): An algorithm which achieves an* $(1+\epsilon)$ *approximation in time* $poly(size, 1/\epsilon)$*, for any* $\epsilon > 0$.

Definition 4.2.3 *PTAS(Polynomial Time Approx. Scheme): Approximation scheme which achieves an* $(1+\epsilon)$ *approximation in time* $poly(size)$ *, for any* $\epsilon > 0$.

Note that an $FPTAS$ is the best algorithm possible for an NP-Hard problem. As mentioned before, knapsack also has a pseudo polytime algorithm. In fact, problems with an $FPTAS$ often have pseudo polytime algorithms. The following theorem formalizes this.

Theorem 4.2.4 *Suppose that an $\mathcal{NP}$ Hard optimization problem has an integral objective function, and the value of the function at its optimal solution is bounded by some polynomial in the size of the problem in unary, then an $FPTAS$ for that problem implies an exact pseudo-polytime algorithm.*

Proof: Suppose that $B = poly(size)$, where *size* is the size of the problem in unary, upper bounds the optimal objective function value. Then we pick $\epsilon = \frac{1}{2B}$ an run the $FPTAS$ with this value. Then

$$ALG \leq OPT \left(1 + \frac{1}{2B}\right) < OPT + 1$$

Since the objective function is integral, $ALG = OPT$, and we obtain an optimal solution. The algorithm runs in time $poly(size)$. ■

References

[1] M. Andrews, L. Zhang. Hardness of the undirected edge-disjoint paths problem. In: *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing (STOC)* (2005), pp: 276–283

[2] C. Chekuri, S. Khanna. Edge disjoint paths revisited. In: *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)* (2003), pp: 628–637

[3] V. Guruswami, S. Khanna, R. Rajaraman, B. Shepherd, M. Yannakakis. Near-optimal hardness results and approximation algorithms for edge-disjoint paths and related problems. In: *Proceedings of thirty-first annual ACM symposium on Theory of computing (STOC)* (19993), pp: 19–28

[4] J. Kleinberg. Approximation Algorithms for Disjoint Paths Problems. *Ph.D Thesis, Dept. of EECS, MIT* (1996).