CS 598CSC: Approximation Algorithms	Lecture date: January 19, 2011
Instructor: Chandra Chekuri	Scribe: Nitish Korula (2009)

Notes edited by instructor in 2011.

1 Introduction/Administrivia

- Course website: `http://www.cs.uiuc.edu/class/sp11/cs598csc/`.
- Recommended books:
  - *The Design of Approximation Algorithms* by David Shmoys and David Williamson, Cambridge University Press, coming in 2011. Free online copy available at `http://www.designofapproxalgs.com/`.
  - *Approximation Algorithms* by Vijay Vazirani, Springer-Verlag, 2004.
- 6 homework sets, the first 3 required. The last 3 sets can be replaced by a project.

Course Objectives

1. To appreciate that not all intractable problems are the same. **NP** optimization problems, identical in terms of exact solvability, can appear very different from the approximation point of view. This sheds light on why, in practice, some optimization problems (such as Knapsack) are easy, while others (like Clique) are extremely difficult.
2. To learn techniques for design and analysis of approximation algorithms, via some fundamental problems.
3. To build a toolkit of broadly applicable algorithms/heuristics that can be used to solve a variety of problems.
4. To understand reductions between optimization problems, and to develop the ability to relate new problems to known ones.

The complexity class **P** contains the set of problems that can be solved in polynomial time. From a theoretical viewpoint, this describes the class of tractable problems, that is, problems that can be solved efficiently. The class **NP** is the set of problems that can be solved in *non-deterministic* polynomial time, or equivalently, problems for which a solution can be *verified* in polynomial time. **NP** contains many interesting problems that often arise in practice, but there is good reason to believe $\mathbf{P} \neq \mathbf{NP}$. That is, it is unlikely that there exist algorithms to solve **NP** optimization problems efficiently, and so we often resort to heuristic methods to solve these problems.

Heuristic approaches include backtrack search and its variants, mathematical programming methods, local seach, genetic algorithms, tabu search, simulated annealing etc. Some methods are guaranteed to find an optimal solution, though they may take exponential time; others are guaranteed to run in polynomial time, though they may not return an optimal solution. Approximation algorithms fall in the latter category; however, though they do not find an optimal solution, we can give guarantees on the *quality* of the solution found.

Approximation Ratio

To give a guarantee on solution quality, one must first define what we mean by the quality of a solution. We discuss this more carefully in the next lecture; for now, note that each *instance* of an optimization problem has a set of feasible solutions. The optimization problems we consider have an **objective function** which assigns a (real/rational) number/value to each feasible solution of each instance I. The goal is to find a feasible solution with minimum objective function value or maximum objective function value. The former problems are minimization problems and the latter are maximization problems.

For each instance I of a problem, let $\mathrm{OPT}(I)$ denote the value of an optimal solution to instance I. We say that an algorithm $\mathcal{A}$ is an α-approximation algorithm for a problem if, for *every* instance I, the value of the feasible solution returned by $\mathcal{A}$ is within a (multiplicative) factor of α of $\mathrm{OPT}(I)$. Equivalently, we say that $\mathcal{A}$ is an approximation algorithm with *approximation ratio* α. For a minimization problem we would have $\alpha \geq 1$ and for a maximization problem we would have $\alpha \leq 1$. However, it is not uncommon to find in the literature a different convention for maximization problems where one says that $\mathcal{A}$ is an α-approximation algorithm if the value of the feasible solution returned by $\mathcal{A}$ is at least $\frac{1}{\alpha} \cdot \mathrm{OPT}(I)$; the reason for using convention is so that approximation ratios for both minimization and maximization problems will be ≥ 1. In this course we will for the most part use the convention that $\alpha \geq 1$ for minimization problems and $\alpha \leq 1$ for maximization problems.

Remarks:

1. The approximation ratio of an algorithm for a minimization problem is the *maximum* (or supremum), over all instances of the problem, of the ratio between the values of solution returned by the algorithm and the optimal solution. Thus, it is a bound on the *worst-case* performance of the algorithm.

2. The approximation ratio α can depend on the size of the instance I, so one should technically write $\alpha(|I|)$.

3. A natural question is whether the approximation ratio should be defined in an *additive* sense. For example, an algorithm has an α-approximation for a minimization problem if it outputs a feasible solution of value at most $\mathrm{OPT}(I) + \alpha$ for all I. This is a valid definition and is the more relevant one in some settings. However, for many **NP** problems it is easy to show that one cannot obtain any interesting additive approximation (unless of course $P = NP$) due to scaling issues. We will illustrate this via an example later.

Pros and cons of the approximation approach:

Some advantages to the approximation approach include:

1. It explains why problems can vary considerably in difficulty.

2. The analysis of problems and problem instances distinguishes easy cases from difficult ones.

3. The worst-case ratio is *robust* in many ways. It allows *reductions* between problems.

4. Algorithmic ideas/tools are valuable in developing heuristics, including many that are practical and effective.

As a bonus, many of the ideas are beautiful and sophisticated, and involve connections to other areas of mathematics and computer science.

Disadvantages include:

1. The focus on *worst-case measures* risks ignoring algorithms or heuristics that are practical or perform well *on average*.
2. Unlike, for example, integer programming, there is often no incremental/continuous tradeoff between the running time and quality of solution.
3. Approximation algorithms are often limited to cleanly stated problems.
4. The framework does not (at least directly) apply to decision problems or those that are inapproximable.

Approximation as a broad lens

The use of approximation algorithms is not restricted solely to **NP**-Hard optimization problems. In general, ideas from approximation can be used to solve many problems where finding an exact solution would require too much of any resource.

A resource we are often concerned with is *time*. Solving **NP**-Hard problems exactly would (to the best of our knowledge) require exponential time, and so we may want to use approximation algorithms. However, for large data sets, even polynomial running time is sometimes unacceptable. As an example, the best exact algorithm known for the MATCHING problem in general graphs requires $O(n^3)$ time; on large graphs, this may be not be practical. In contrast, a simple greedy algorithm takes near-linear time and outputs a matching of cardinality at least $1/2$ that of the maximum matching; moreover there have been randomized sub-linear time algorithms as well.

Another often limited resource is *space*. In the area of data streams/streaming algorithms, we are often only allowed to read the input in a single pass, and given a small amount of additional storage space. Consider a network switch that wishes to compute statistics about the packets that pass through it. It is easy to exactly compute the average packet length, but one cannot compute the median length exactly. Surprisingly, though, many statistics can be approximately computed.

Other resources include programmer time (as for the MATCHING problem, the exact algorithm may be significantly more complex than one that returns an approximate solution), or communication requirements (for instance, if the computation is occurring across multiple locations).

2 The Steiner Tree Problem

In the STEINER TREE problem, the input is a graph $G(V, E)$, together with a set of *terminals* $S \subseteq V$, and a cost $c(e)$ for each edge $e \in E$. The goal is to find a minimum-cost tree that connects all terminals, where the cost of a subgraph is the sum of the costs of its edges.

The STEINER TREE problem is **NP**-Hard, and also **APX**-Hard [2]. The latter means that there is a constant $\delta > 1$ such that it is **NP**-Hard to approximate the solution to within a ratio of less than δ; it is currently known that it is hard to approximate the STEINER TREE problem to within a ratio of $95/94$ [5].[1]

[1] Variants of the STEINER TREE problem, named after Jakob Steiner, have been studied by Fermat, Weber, and others for centuries. The front cover of the course textbook contains a reproduction of a letter from Gauss to Schumacher on a Steiner tree question.

Note: If $|S| = 2$ (that is, there are only 2 terminals), an optimal Steiner Tree is simply a shortest path between these 2 terminals. If $S = V$ (that is, all vertices are terminals), an optimal solution is simply a minimum spanning tree of the input graph. In both these cases, the problem can be solved exactly in polynomial time.

Question: Can you find an efficient algorithm to exactly solve the Steiner Tree problem with 3 terminals? How about the case when $|S|$ is a fixed constant?

Observe that to solve the STEINER TREE problem on a graph G, it suffices to solve it on the metric completion of G, defined below. (Why is this true?)

Definition: Given a connected graph $G(V, E)$ with edge costs, the *metric completion* of G is a complete graph $H(V, E')$ such that for each $u, v \in V$, the cost of edge uv in H is the cost of the shortest path in G from u to v.

The graph H with edge costs is *metric*, because the edge costs satisfy the triangle inequality: $\forall u, v, w, \quad cost(uv) \leq cost(uw) + cost(wv)$.

Figure 1: On the left, a graph. On the right, its metric completion, with new edges and modified edge costs in red.

We now look at two approximation algorithms for the STEINER TREE problem.

2.1 The MST Algorithm

The following algorithm, with an approximation ratio of $(2 - 2/|S|)$ is due to [12]:

STEINERMST$(G(V, E), S \subseteq V)$:
Let $H(V, E') \leftarrow$ metric completion of G.
Let $T \leftarrow$ MST of $H[S]$.
Output T.

(Here, we use the notation $H[S]$ to denote the subgraph of H induced by the set of terminals S.)

The following lemma is central to the analysis of the algorithm STEINERMST.

Lemma 1 *For any instance I of* STEINER TREE*, let H denote the metric completion of the graph, and S the set of terminals. There exists a spanning tree in $H[S]$ (the graph induced by terminals) of cost at most $2(1 - \frac{1}{|S|})$OPT, where* OPT *is the cost of an optimal solution to instance I.*

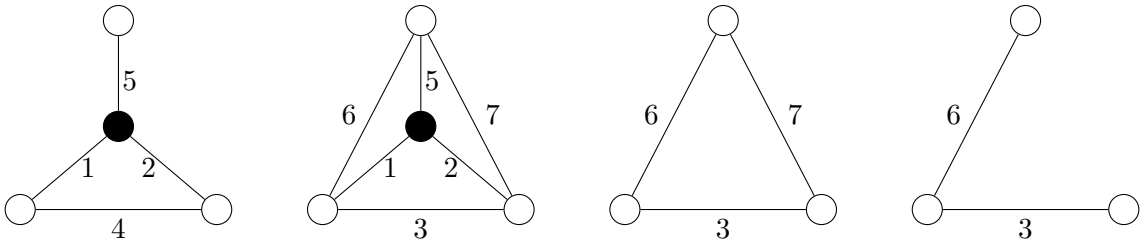


Figure 2: Illustrating the MST Heuristic for Steiner Tree

Before we prove the lemma, we note that if there exists *some spanning tree* in $H[S]$ of cost at most $2(1-\frac{1}{|S|})\text{OPT}$, the minimum spanning tree has at most this cost. Therefore, Lemma 1 implies that the algorithm SteinerMST is a $2(1-\frac{1}{|S|})$-approximation for the Steiner Tree problem.

Proof of Lemma 1. Let T^* denote an optimal solution in H to the given instance, with cost $c(T^*)$. Double all the edges of T^* to obtain an Eulerian graph, and fix an Eulerian Tour W of this graph. (See Fig. 3 above.) Now, shortcut edges of W to obtain a tour W' of the vertices in T^* in which each vertex is visited exactly once. Again, shortcut edges of W' to eliminate all non-terminals; this gives a walk W'' that visits each terminal exactly once.

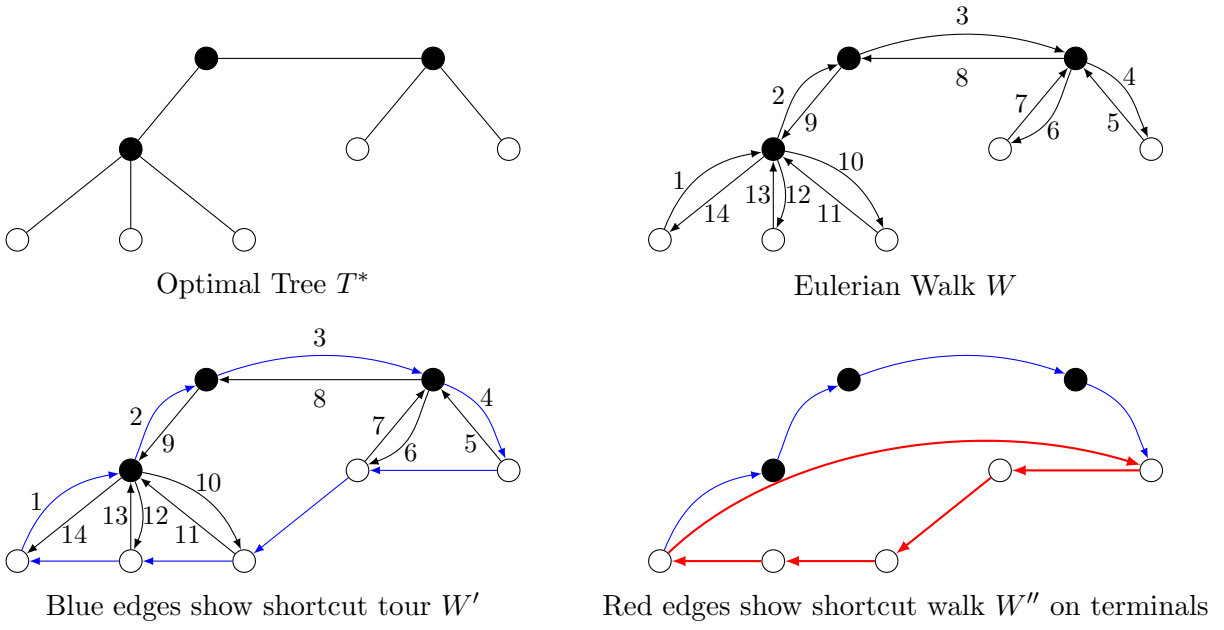
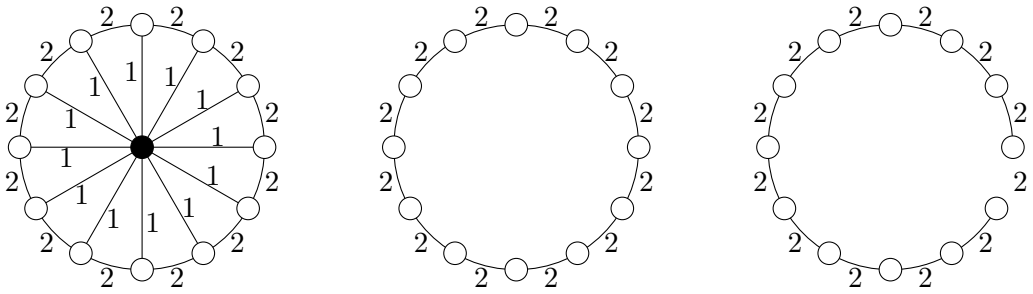


Figure 3: Doubling edges of T^* and shortcutting gives a low-cost spanning tree on terminals.

It is easy to see that $c(W'') \leq c(W') \leq c(W) = 2c(T^*)$, where the inequalities follow from the fact that by shortcutting, we can only decrease the length of the walk. (Recall that we are working in the metric completion H.) Now, delete the heaviest edge of W'' to obtain a path through all the terminals in S, of cost at most $(1-\frac{1}{|S|})c(W'')$. This path is a spanning tree of the terminals, and contains *only* terminals; therefore, there exists a spanning tree in $H[S]$ of cost at most $2(1-\frac{1}{|S|})c(T^*)$. □

A tight example: The following example (Fig. 4 below) shows that this analysis is tight; there are instances of STEINER TREE where the STEINERMST algorithm finds a tree of cost $2(1-\frac{1}{S})$OPT. Here, each pair of terminals is connected by an edge of cost 2, and each terminal is connected to the central non-terminal by an edge of cost 1. The optimal tree is a star containing the central non-terminal, with edges to all the terminals; it has cost $|S|$. However, the only trees in $H[S]$ are formed by taking $|S|-1$ edges of cost 2; they have cost $2(|S|-1)$.



Graph G; not all edges shown $H[S]$; not all edges shown. An MST of $H[S]$.

Figure 4: A tight example for the STEINERMST algorithm

2.2 The Greedy/Online Algorithm

We now describe another simple algorithm for the STEINER TREE problem, due to [9]:

```
GREEDYSTEINER(G(V,E), S ⊆ V):
Let {s_1, s_2, ... s_|S|} be an arbitrary ordering of the terminals.
Let T ← {s_1}
For (i from 2 to |S|):
   Let P_i be the shortest path in G from s_i to T.
   Add P_i to T.
```

GREEDYSTEINER is a $\lceil \log_2 |S| \rceil$-approximation algorithm; here, we prove a slightly weaker result.

Theorem 2 *The algorithm* GREEDYSTEINER *has an approximation ratio of* $2H_{|S|} \approx 2\ln|S|$*, where* $H_i = \sum_{j=1}^{i} 1/j$ *denotes the i'th harmonic number.*

Note that this is an *online* algorithm; terminals are considered in an arbitrary order, and when a terminal is considered, it is immediately connected to the existing tree. Thus, even if the algorithm *could not see the entire input at once*, but instead terminals were revealed one at a time and the algorithm had to produce a Steiner tree at each stage, the algorithm GREEDYSTEINER outputs a tree of cost no more than $O(\log|S|)$ times the cost of the optimal tree.

To prove Theorem 2, we introduce some notation. Let $c(i)$ denote the cost of the path P_i used in the ith iteration to connect the terminal s_i to the already existing tree. Clearly, the total cost of the tree is $\sum_{i=1}^{|S|} c(i)$. Now, let $\{i_1, i_2, \ldots i_{|S|}\}$ be a permutation of $\{1, 2, \ldots |S|\}$ such that $c(i_1) \geq c(i_2) \geq \ldots \geq c(i_{|S|})$. (That is, relabel the terminals in decreasing order of the cost paid to connect them to the tree that exists when they are considered by the algorithm.)

Claim 3 *For all j, the cost $c(i_j)$ is at most $2\mathrm{OPT}/j$, where OPT is the cost of an optimal solution to the given instance.*

Proof: Suppose by way of contradiction this were not true; since s_{i_j} is the terminal with jth highest cost of connection, there must be j terminals that each pay more than $2\mathrm{OPT}/j$ to connect to the tree that exists when they are considered. Let $S' = \{s_{i_1}, s_{i_2}, \ldots s_{i_j}\}$ denote this set of terminals.

We argue that no two terminals in $S' \cup \{s_1\}$ are within distance $2\mathrm{OPT}/j$ of each other. If some pair x, y were within this distance, one of these terminals (say y) must be considered later by the algorithm than the other. But then the cost of connecting y to the already existing tree (which includes x) must be at most $2\mathrm{OPT}/j$, and we have a contradiction.

Therefore, the minimum distance between any two terminals in $S' \cup \{s_1\}$ must be greater than $2\mathrm{OPT}/j$. Since there must be j edges in any MST of these terminals, an MST must have cost greater than $2\mathrm{OPT}$. But the MST of a subset of terminals cannot have cost more than $2\mathrm{OPT}$, exactly as argued in the proof of Lemma 1. Therefore, we obtain a contradiction. □

Given this claim, it is easy to prove Theorem 2.

$$\sum_{i=1}^{|S|} c(i) = \sum_{j=1}^{|S|} c(i_j) \leq \sum_{j=1}^{|S|} \frac{2\mathrm{OPT}}{j} = 2\mathrm{OPT} \sum_{j=1}^{|S|} \frac{1}{j} = 2H_{|S|} \cdot \mathrm{OPT}.$$

Question: Give an example of a graph and an ordering of terminals such that the output of the Greedy algorithm is $\Omega(\log |S|)\mathrm{OPT}$.

Remark: We emphasize again that the analysi above holds for *every* ordering of the terminals. A natural variant might be to adaptively order the terminals so that in each iteration i , the algorithm picks the terminal s_i to be the one closest to the already existing tree T built in the first i iterations. Do you see that this is equivalent to using the MST Heuristic with Prim's algorithm for MST? This illustrates the need to be careful in the design and analysis of heuristics.

Other Results on Steiner Trees

The 2-approximation algorithm using the MST Heuristic is not the best approximation algorithm for the Steiner Tree problem currently known. Some other results on this problem are listed below.

1. The first algorithm to obtain a ratio of better than 2 was due to due to Alexander Zelikovsky [13]; the approximation ratio of this algorithm was $11/6 \approx 1.83$. Until very recently the best known approximation ratio in general graphs is $1 + \frac{\ln 3}{2} \approx 1.55$; this is achieved by an algorithm of [11] that combines the MST heuristic with Local Search. Very recently Byrka *et al.* gave an algorithm with an approximation ratio of 1.39 [4] which is currently the best known for this problem.

2. The *bidirected cut* LP relaxation for the Steiner Tree was proposed by [7]; it has an integrality gap of at most $2(1 - \frac{1}{|S|})$, but it is conjectured that the gap is smaller. No algorithm is currently known that exploits this LP relaxation to obtain an approximation ratio better than that of the SteinerMST algorithm. Though the true integrality gap is not known, there are examples due to [8, 10] that show it is at least $8/7 \approx 1.14$.

3. For many applications, the vertices can be modeled as points on the plane, where the distance between them is simply the Euclidean distance. The MST-based algorithm performs fairly well on such instances; it has an approximation ratio of $2/\sqrt{3} \approx 1.15$ [6]. An example which achieves this bound is three points at the corners of an equilateral triangle, say of side-length 1; the MST heuristic outputs a tree of cost 2 (two sides of the triangle) while the optimum solution is to connect the three points to a Steiner vertex which is the circumcenter of the triangle. One can do better still for instances in the plane (or in any Euclidean space of small-dimensions); for any $\epsilon > 0$, there is a $1 + \epsilon$-approximation algorithm that runs in polynomial time [1]. Such an approximation scheme is also known for planar graphs [3] and more generally bounded-genus graphs.

References

[1] S. Arora. Polynomial-time Approximation Schemes for Euclidean TSP and other Geometric Problems. *J. of the ACM* 45(5): 753–782, 1998.

[2] M. Bern and P. Plassmann. The Steiner problems with edge lengths 1 and 2. *Information Processing Letters* 32, 171–176, 1989.

[3] G. Borradaile, C. Mathieu, and P. Klein. A polynomial-time approximation scheme for Steiner tree in planar graphs. *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2007, pp. 1285–1294.

[4] J. Byrka, F. Grandoni, T. Rothvoss, and L. Sanita. An improved LP-based approximation for steiner tree. *Proc. of ACM Symposium on Theory of Computing (STOC)*, 583–592, 2010.

[5] M. Chlebik and J. Chlebikova. Approximation Hardness of the Steiner Tree Problem on Graphs. *Proc. 8th Scandinavian Workshop on Algorithm Theory*, Springer-Verlag, 170–179, 2002.

[6] D. Z. Du and F. K. Hwang. A proof of Gilbert-Pollak's conjecture on the Steiner ratio. *Algorithmica*, 7: 121–135, 1992.

[7] J. Edmonds. Optimum branchings. *J. of Research of the National Bureau of Standards, Section B*, 71:233–240, 1967.

[8] M. Goemans. Unpublished Manuscript, 1996.

[9] M. Imaze and B.M. Waxman. Dynamic Steiner tree problem. *SIAM J. on Discrete Math.*, 4(3): 369–184, 1991.

[10] J. Konemann, D. Pritchard, and K. Tan. A partition based relaxation for Steiner trees. Manuscript, 2007. `arXiv:0712.3568`, `http://arxiv.org/abs/0712.3568v1`

[11] G. Robins and A. Zelikovsky. Tighter Bounds for Graph Steiner Tree Approximation. *SIAM J. on Discrete Math.*, 19(1): 122–134, 2005.

[12] J. Takahashi and A. Matsuyama. An approximate solution for the Steiner problem in graphs. *Math. Jap.*, 24: 573–577, 1980.

[13] A. Zelikovsky. An 11/6-approximation algorithm for the network Steiner problem. *Algorithmica*, 9: 463–470, 1993.

CS 598CSC: Approximation Algorithms	Lecture date: January 21, 2011
Instructor: Chandra Chekuri	Scribe: Sungjin Im (2009)

Notes edited by instructor in 2011.

In the previous lecture, we had a quick overview of several basic aspects of approximation algorithms. We also addressed approximation (both offline and online) algorithms for the Steiner Tree Problem. In this lecture, we explore another important problem – the Traveling Salesperson Problem (TSP).

1 The Traveling Salesperson Problem (TSP)

1.1 TSP in Undirected Graphs

In the Traveling Salesperson Problem (TSP), we are given an undirected graph $G = (V, E)$ and cost $c(e) > 0$ for each edge $e \in E$. Our goal is to find a Hamiltonian cycle with minimum cost. A cycle is said to be Hamiltonian if it visits every vertex in V exactly once.

TSP is known to be NP-Hard. Moreover, we cannot hope to find a good approximation algorithm for it unless $P = NP$. This is because if one can give a good approximation solution to TSP in polynomial time, then we can exactly solve the NP-Complete Hamiltonian cycle problem (HAM) in polynomial time, which is not possible unless $P = NP$. Recall that HAM is a decision problem: given a graph $G = (V, E)$, does G have a Hamiltonian cycle?

Theorem 1 ([3]) *Let $\alpha : \mathbb{N} \to \mathbb{N}$ be a polynomial-time computable function. Unless $P = NP$ there is no polynomial-time algorithm that on every instance I of TSP outputs a solution of cost at most $\alpha(|I|) \cdot \mathrm{OPT}(I)$.*

Proof: For the sake of contradiction, suppose we have an approximation algorithm $\mathcal{A}$ for TSP with an approximation ratio $\alpha(|I|)$. We show a contradiction by showing that using $\mathcal{A}$, we can exactly solve HAM in polynomial time. Let $G = (V, E)$ be the given instance of HAM. We create a new graph $H = (V, E')$ with cost $c(e)$ for each $e \in E'$ such that $c(e) = 1$ if $e \in E$, otherwise $c(e) = B$, where $B = n\alpha(n) + 2$ and $n = |V|$. Note that this is a polynomial-time reduction since α is a polynomial-time computable function.

We observe that if G has a Hamiltonian cycle, $\mathrm{OPT} = n$, otherwise $\mathrm{OPT} \geq n - 1 + B \geq n\alpha(n) + 1$. (Here, OPT denotes the cost of an optimal TSP solution in H.) Note that there is a "gap" between when G has a Hamiltonian cycle and when it does not. Thus, if $\mathcal{A}$ has an approximation ratio of $\alpha(n)$, we can tell whether G has a Hamiltonian cycle or not: Simply run $\mathcal{A}$ on the graph H; if $\mathcal{A}$ returns a TSP tour in H of cost at most $\alpha(n)n$ output that G has a Hamilton cycle, otherwise output that G has no Hamilton cycle. We leave it as an exercise to formally verify that this would solve HAM in polynomial time. □

Since we cannot even approximate the general TSP problem, we consider more tractable variants.

- *Metric-TSP*: In Metric-TSP, the instance is a complete graph $G = (V, E)$ with cost $c(e)$ on $e \in E$, where c satisfies the triangle inequality, i.e. $c(uw) \leq c(uv) + c(vw)$ for any $u, v, w \in V$.

- *TSP-R*: TSP with repetitions of vertices allowed. The input is a graph $G = (V, E)$ with non-negative edge costs as in TSP. Now we seek a minimum-cost *walk* that visits each vertex at least once and returns to the starting vertex.

Exercise: Show that an α-approximation for Metric-TSP implies an α-approximation for TSP-R and vice-versa.

We focus on Metric-TSP for the rest of this section. We first consider a natural greedy approach, the Nearest Neighbor Heuristic (NNH).

NEAREST NEIGHBOR HEURISTIC($G(V, E), c : E \to \mathcal{R}^+$):
Start at an arbitrary vertex s,
While (there are unvisited vertices)
  From the current vertex u, go to the nearest unvisited vertex v.
Return to s.

Exercise:

1. Prove that NNH is an $O(\log n)$-approximation algorithm. (**Hint:** Think back to the proof of the $2H_{|S|}$-approximation for the Greedy Steiner Tree Algorithm.)
2. NNH is not an $O(1)$-approximation algorithm; can you find an example to show this? In fact one can show a lower bound of $\Omega(\log n)$ on the approximation-ratio achieved by NNH.

There are constant-factor approximation algorithms for TSP; we now consider an MST-based algorithm. See Fig 1.

TSP-MST($G(V, E), c : E \to \mathcal{R}^+$):
Compute an MST T of G.
Obtain an Eulerian graph $H = 2T$ by *doubling* edges of T
An Eulerian tour of $2T$ gives a tour in G.
Obtain a Hamiltonian cycle by shortcutting the tour.

Theorem 2 *MST heuristic(TSP-MST) is a 2-approximation algorithm.*

Proof: We have $c(T) = \sum_{e \in E(T)} c(e) \leq \text{OPT}$, since we can get a spanning tree in G by removing any edge from the optimal Hamiltonian cycle, and T is a MST. Thus $c(H) = 2c(T) \leq 2\text{OPT}$. Also shortcutting only decreases the cost. □

We observe that the loss of a factor 2 in the approximation ratio is due to doubling edges; we did this in order to obtain an Eulerian tour. But any graph in which all vertices have even degree is Eulerian, so one can still get an Eulerian tour by adding edges only between odd degree vertices in T. Christofides Heuristic [2] exploits this and improves the approximation ratio from 2 to 3/2. See Fig 2 for a snapshot.

CHRISTOFIDES HEURISTIC($G(V, E), c : E \to \mathcal{R}^+$):
Compute an MST T of G.
Let S be the vertices of odd degree in T. (Note: $|S|$ is even)
Find a minimum cost matching M on S in G
Add M to T to obtain an Eulerian graph H.
Compute an Eulerian tour of H.
Obtain a Hamilton cycle by shortcutting the tour.

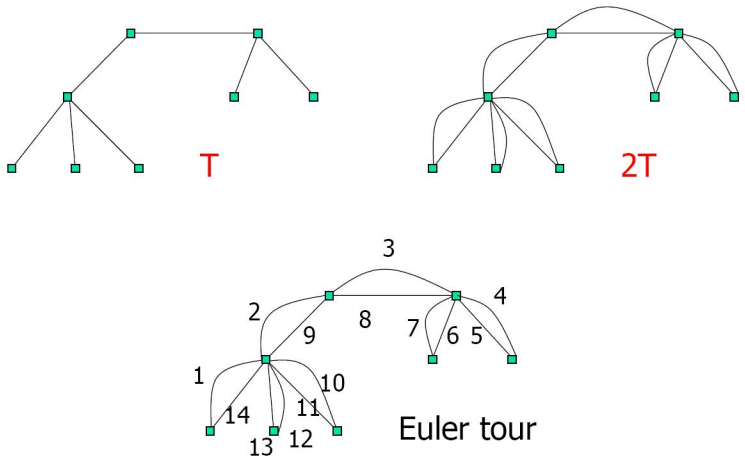


Figure 1: MST Based Heuristic

Theorem 3 *Christofides Heuristic is a 1.5-approximation algorithm.*

Proof: The main part of the proof is to show that $c(M) \leq .5\text{OPT}$. Suppose that $c(M) \leq .5\text{OPT}$. Then, since the solution of Christofides Heuristic is obtained by shortcutting the Eulerian tour on H, its cost is no more than $c(H) = c(T) + c(M) \leq 1.5\text{OPT}$. (Refer to the proof of Lemma 2 for the fact $c(T) \leq \text{OPT}$.) Therefore we focus on proving that $c(M) \leq .5\text{OPT}$.

Let F^* be an optimal tour in G of cost OPT; since we have a metric-instance we can assume without loss of generality that F^* is a Hamiltonian cycle. We obtain a Hamiltonian cycle F^*_S in the graph $G[S]$ by short-cutting the portions of F^* that touch the vertices $V \setminus S$. By the metric-condition, $c(F^*_S) \leq c(F^*) = \text{OPT}$. Let $S = \{v_1, v_2, \ldots, v_{|S|}\}$. Without loss of generality F^*_S visits the vertices of S in the order $v_1, v_2, \ldots, v_{|S|}$. Recall that $|S|$ is even. Let $M_1 = \{v_1v_2, v_3v_4, ...v_{|S|-1}v_{|S|}\}$ and $M_2 = \{v_2v_3, v_4v_5, ...v_{|S|}v_1\}$. Note that both M_1 and M_2 are matchings, and $c(M_1) + c(M_2) = c(F^*_S) \leq \text{OPT}$. We can assume without loss of generality that $c(M_1) \leq c(M_2)$. Then we have $c(M_1) \leq .5\text{OPT}$. Also we know that $c(M) \leq c(M_1)$, since M is a minimum cost matching on S in $G[S]$. Hence we have $c(M) \leq c(M_1) \leq .5\text{OPT}$, which completes the proof. □

Notes:

1. In practice, local search heuristics are widely used and they perform extremely well. A popular heuristic *2-Opt* is to swap pairs from xy, zw to xz, yw or xw, yz, if it improves the tour.

2. There have been no improvements to Metric-TSP since Christofides heuristic was discovered in 1976. It remains a major open problem to improve the approximation ratio of $\frac{3}{2}$ for Metric-TSP; it is conjectured that the Held-Karp LP relaxation [4] gives a ratio of $\frac{4}{3}$. Very recently a breakthrough has been announced by Oveis-Gharan, Saberi and Singh who claim a $3/2 - \delta$ approximation for some small but fixed $\delta > 0$ for the important special case where $c(e) = 1$ for each edge e.

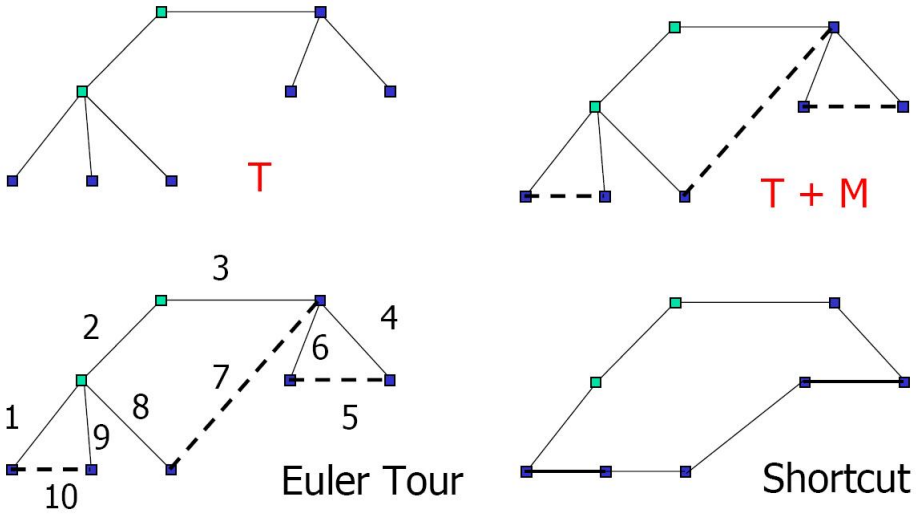


Figure 2: Christofides Heuristic

1.2 TSP in Directed Graphs

In this subsection, we consider TSP in directed graphs. As in undirected TSP, we need to relax the problem conditions to get any positive result. Again, allowing each vertex to be visited multiple times is equivalent to imposing the asymmetric triangle inequality $c(u, w) \leq c(u, v) + c(v, w)$ for all u, v, w. This is called the asymmetric TSP (ATSP) problem. We are given a directed graph $G = (V, A)$ with cost $c(a) > 0$ for each arc $a \in A$ and our goal is to find a closed walk visiting all vertices. Note that we are allowed to visit each vertex multiple times, as we are looking for a walk, not a cycle. For an example of a valid Hamiltonian walk, see Fig 3.

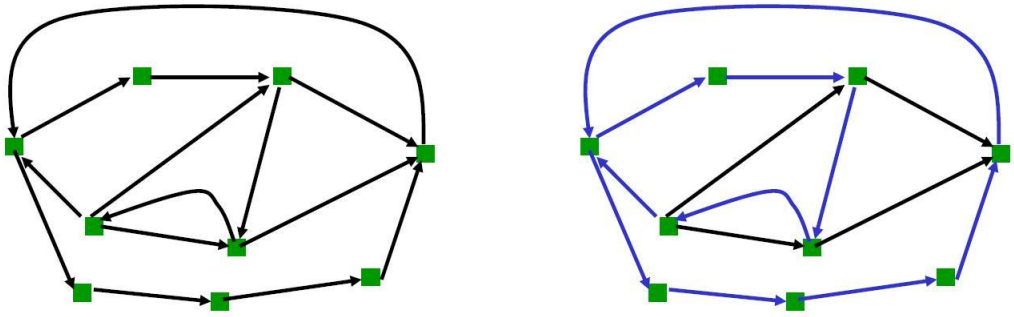


Figure 3: A directed graph and a valid Hamiltonian walk

The MST-based heuristic for the undirected case has no meaningful generalization to the directed setting This is because costs on edges are not symmetric. Hence, we need another approach. The *Cycle Shrinking Algorithm* repeatedly finds a min-cost cycle cover and shrinks cycles, combining the cycle covers found. Recall that a cycle cover is a collection of disjoint cycles covering all vertices. It is known that finding a minimum-cost cycle cover can be done in polynomial time (see

Homework 0). The Cycle Shrinking Algorithm achieves a $\log_2 n$ approximation ratio.

CYCLE SHRINKING ALGORITHM$(G(V,A), c : A \to \mathcal{R}^+)$:
Transform G s.t. G is complete and satisfies $c(u,v) + c(v,w) \geq c(u,w)$ for $\forall u, v, w$
If $\lvert V \rvert = 1$ output the trivial cycle consisting of the single node
Find a minimum cost cycle cover with cycles $C_1, \ldots, C_k$
From each C_i pick an arbitrary proxy node v_i
Recursively solve problem on $G[\{v_1, \ldots, v_k\}]$ to obtain a solution C
$C' = C \cup C_1 \cup C_2 \ldots C_k$ is a Eulerian graph.
Shortcut C' to obtain a cycle on V and output C'.

For a snapshot of the Cycle Shrinking Algorithm, see Fig 4.

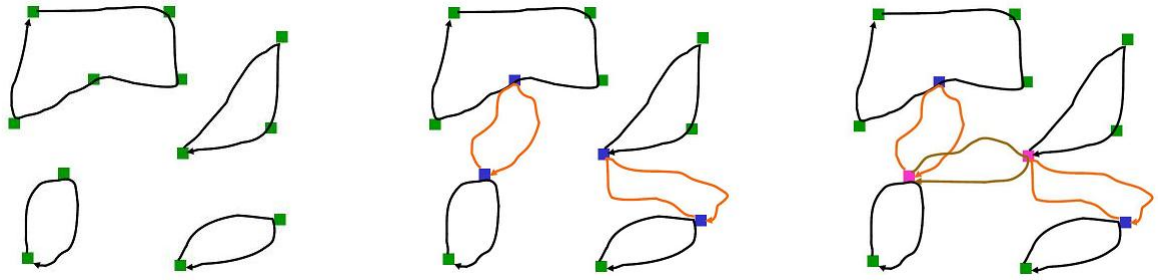


Figure 4: A snapshot of Cycle Shrinking Algorithm. To the left, a cycle cover $\mathcal{C}_1$. In the center, blue vertices indicate proxy nodes, and a cycle cover $\mathcal{C}_2$ is found on the proxy nodes. To the right, pink vertices are new proxy nodes, and a cycle cover $\mathcal{C}_3$ is found on the new proxy nodes.

Lemma 4 *Let the cost of edges in G satisfy the asymmetric triangle inequality. Then for any $S \subseteq V$, the cost of an optimal TSP tour in $G[S]$ is at most the cost of an optimal TSP tour in G.*

Proof: Since G satisfies the triangle inequality there is an optimal tour TSP tour in G that is a Hamiltonian cycle C. Given any $S \subseteq V$ the cycle C can be short-cut to produce another cycle C' that visits only S and whose cost is at most the cost of C. □

Lemma 5 *The cost of a min-cost cycle-cover is at most the cost of an optimal TSP tour.*

Proof: An optimal TSP tour is a cycle cover. □

Theorem 6 *The Cycle Shrinking Algorithm is a $\log_2 n$-approximation for ATSP.*

Proof: We prove the above by induction on n the number of nodes in G. It is easy to see that the algorithm finds an optimal solution if $n \leq 2$. The main observation is that the number of cycles in the cycle-cover is at most $\lfloor n/2 \rfloor$; this follows from the fact that each cycle in the cover has to have at least 2 nodes and they are disjoint. Thus $k \leq \lfloor n/2 \rfloor$. Let $\mathrm{OPT}(S)$ denote the cost of an optimal solution in $G[S]$. From Lemma 4 we have that $\mathrm{OPT}(S) \leq \mathrm{OPT}(V) = \mathrm{OPT}$ for all $S \subseteq V$. The algorithm recurses on the proxy nodes $S = \{v_1, \ldots, v_k\}$. Note that $|S| < n$, and by induction, the cost of the cycle C output by the recursive call is at most $(\log_2 |S|)\mathrm{OPT}(S) \leq (\log_2 |S|)\mathrm{OPT}$.

The algorithm outputs C' whose cost is at most the cost of C plus the cost of the cycle-cover computed in G. The cost of the cycle cover is at most OPT (Lemma 5). Hence the cost of C' is at most $(\log_2 |S|)\mathrm{OPT} + \mathrm{OPT} \leq (\log_2 n/2)\mathrm{OPT} + \mathrm{OPT} \leq (\log_2 n)\mathrm{OPT}$; this finishes the inductive proof. □

Notes:

1. The running time of the Cycle Shrinking Algorithm is $O(T(n))$ where $T(n)$ is the time to find a min-cost cycle cover (why?). In Homework 0 you have a problem that reduces this problem to that of finding a min-cost perfect matching in an *undirected* graph. This can be done in $O(n^3)$-time. One can improve the running time to $O(n^2)$ for by approximating the min-cost cycle-cover problem; one loses an additional constant factor.

2. It has remained an open problem for more than 25 years whether there exists a constant factor approximation for ATSP. Recently Asadpour *et al.* [1] have obtained an $O(\log n / \log \log n)$-approximation for ATSP using some very novel ideas and a well-known LP relaxation.

2 Some Definitions

2.1 NP Optimization Problems

In this section, we cover some formal definitions related to approximation algorithms. We start from the definition of optimization problems. A problem is simply an infinite collection of *instances*. Let Π be an optimization problem. Π can be either a minimization or maximixation problem. Instances I of Π are a subset of Σ^* where Σ is a finite encoding alphabet. For each instance I there is a set of feasible solutions $\mathcal{S}(I)$. We restrict our attention to real/rational-valued optimization problems; in these problems each feasible solution $S \in \mathcal{S}(I)$ has a value $val(S, I)$. For a minimization problem Π the goal is, given I, find $\mathrm{OPT}(I) = \min_{S \in \mathcal{S}(I)} val(S, I)$.

Now let us formally define NP optimization (NPO) which is the class of optimization problems corresponding to NP.

Definition 7 *Π is in NPO if*

- *Given $x \in \Sigma^*$, there is a polynomial-time algorithm that decide if x is a valid instance of Π. That is, we can efficiently check if the input string is well-formed. This is a basic requirement that is often not spelled out.*
- *For each I, and $S \in \mathcal{S}(I)$, $|S| \leq poly(|I|)$. That is, the solution are of size polynomial in the input size.*
- *There exists a poly-time decision procedure that for each I and $S \in \Sigma^*$, decides if $S \in \mathcal{S}(I)$. This is the key property of NP; we should be able to* verify *solutions efficiently.*
- *$val(I, S)$ is a polynomial-time computable function.*

We observe that for a minimization NPO problem Π, there is a associated natural decision problem $L(\Pi) = \{(I, B) : \mathrm{OPT}(I) \leq B\}$ which is the following: given instance I of Π and a number B, is the optimal value on I at most B? For maximization problem Π we reverse the inequality in the definition.

Lemma 8 *$L(\Pi)$ is in NP if Π is in NPO.*

2.2 Relative Approximation

When Π is a minimization problem, recall that we say an approximation algorithm $\mathcal{A}$ is said to have approximation ratio α iff

- $\mathcal{A}$ is a polynomial time algorithm
- for all instance I of Π, $\mathcal{A}$ produces a feasible solution $\mathcal{A}(I)$ s.t. $val(\mathcal{A}(I), I) \leq \alpha\, val\,(\mathrm{OPT}(I), I)$. (Note that $\alpha \geq 1$.)

Approximation algorithms for maximization problems are defined similarly. An approximation algorithm $\mathcal{A}$ is said to have approximation ratio α iff

- $\mathcal{A}$ is a polynomial time algorithm
- for all instance I of Π, $\mathcal{A}$ produces a feasible solution $\mathcal{A}(I)$ s.t. $val(\mathcal{A}(I), I) \geq \alpha\, val\,(\mathrm{OPT}(I), I)$. (Note that $\alpha \leq 1$.)

For maximization problems, it is also common to see use $1/\alpha$ (which must be ≥ 1) as approximation ratio.

2.3 Additive Approximation

Note that all the definitions above are about relative approximations; one could also define *additive* approximations. $\mathcal{A}$ is said to be an α-additive approximation algorithm, if for all I, $val(\mathcal{A}(I)) \leq \mathrm{OPT}(I)+\alpha$. Most NPO problems, however, do not allow any additive approximation ratio because $\mathrm{OPT}(I)$ has a scaling property.

To illustrate the scaling property, let us consider Metric-TSP. Given an instance I, let I_β denote the instance obtained by increasing all edge costs by a factor of β. It is easy to observe that for each $S \in \mathcal{S}(I) = \mathcal{S}(I_\beta)$, $val(S, I_\beta) = \beta val(S, I_\beta)$ and $\mathrm{OPT}(I_\beta) = \beta\mathrm{OPT}(I)$. Intuitively, scaling edge by a factor of β scales the value by the same factor β. Thus by choosing β sufficiently large, we can essentially make the additive approximation(or error) negligible.

Lemma 9 *Metric-TSP does not admit an α additive approximation algorithm for any polynomial-time computable α unless $P = NP$.*

Proof: For simplicity, suppose every edge has integer cost. For the sake of contradiction, suppose there exists an additive α approximation $\mathcal{A}$ for Metric-TSP. Given I, we run the algorithm on I_β and let S be the solution, where $\beta = 2\alpha$. We claim that S is the optimal solution for I. We have $val(S, I) = val(S, I_\beta)/\beta \leq \mathrm{OPT}(I_\beta)/\beta + \alpha/\beta = \mathrm{OPT}(I) + 1/2$, as $\mathcal{A}$ is α-additive approximation. Thus we conclude that $\mathrm{OPT}(I) = val(S, I)$, since $\mathrm{OPT}(I) \leq val(S, I)$, and $\mathrm{OPT}(I)$, $val(S, I)$ are integers. This is impossible unless $P = NP$. □

Now let us consider two problems which allow additive approximations. In the Planar Graph Coloring, we are given a planar graph $G = (V, E)$. We are asked to color all vertices of the given graph G such that for any $vw \in E$, v and w have different colors. The goal is to minimize the number of different colors. It is known that to decide if a planar graph admits 3-coloring is NP-complete, while one can always color any planar graph G with using 4 colors. Further, one can efficiently check whether a graph is 2-colorable (that is, if it is bipartite). Thus, the following algorithm is a 1-additive approximation for Planar Graph Coloring: If the graph is bipartite, color it with 2 colors; otherwise, color with 4 colors.

As a second example, consider the Edge Coloring Problem, in which we are asked to color edges of a given graph G with the minimum number of different colors so that no two adjacent edges have different colors. By Vizing's theorem [6], we know that one can color edges with either $\Delta(G)$ or $\Delta(G)+1$ different colors, where $\Delta(G)$ is the maximum degree of G. Since $\Delta(G)$ is a trivial lower bound on the minimum number, we can say that the Edge Coloring Problem allows a 1-additive approximation. Note that it is known to be NP-complete to decide whether the exact minimum number is $\Delta(G)$ or $\Delta(G)+1$.

2.4 Hardness of Approximation

Now we move to hardness of approximation.

Definition 10 (Approximability Threshold) *Given a minimization optimization problem Π, it is said that Π has an approximation threshold $\alpha^*(\Pi)$, if for any $\epsilon > 0$, Π admits a $\alpha^*(\Pi)+\epsilon$ approximation but if it admits a $\alpha^*(\Pi)-\epsilon$ approximation then $P = NP$.*

If $\alpha^*(\Pi) = 1$, it implies that Π is solvable in polynomial time. Many NPO problems Π are known to have $\alpha^*(\Pi) > 1$ assuming that $P \neq NP$. We can say that approximation algorithms try to decrease the upper bound on $\alpha^*(\Pi)$, while hardness of approximation attempts to increase lower bounds on $\alpha^*(\Pi)$.

To prove hardness results on NPO problems in terms of approximation, there are largely two approaches; a direct way by reduction from NP-complete problems and an indirect way via gap reductions. Here let us take a quick look at an example using a reduction from an NP-complete problem.

In the (metric) k-center problem, we are given an undirected graph $G = (V, E)$ and an integer k. We are asked to choose a subset of k vertices from V called centers. The goal is to minimize the maximum distance to a center, i.e. $\min_{S \subseteq V, |S|=k} \max_{v \in V} \mathrm{dist}_G(v, S)$, where $\mathrm{dist}_G(v, S) = \min_{u \in S} \mathrm{dist}_G(u, v)$.

The k-center problem has approximation threshold 2, since there are a few 2-approximation algorithms for k-center and there is no $2-\epsilon$ approximation algorithm for any $\epsilon > 0$ unless $P = NP$. We can prove the inapproximability using a reduction from the decision version of Dominating Set: Given an undirected graph $G = (V, E)$ and an integer k, does G have a dominating set of size at most k? A set $S \subseteq V$ is said to be a dominating set in G if for all $v \in V$, $v \in S$ or v is adjacent to some u in S. Dominating Set is known to be NP-complete.

Theorem 11 ([5]) *Unless $P = NP$, there is no $2-\epsilon$ approximation for k-center for any fixed $\epsilon > 0$.*

Proof: Let I be an instance of Dominating Set Problem consisting of graph $G = (V, E)$ and integer k. We create an instance I' of k-center while keeping graph G and k the same. If I has a dominating set of size k then $\mathrm{OPT}(I') = 1$, since every vertex can be reachable from the Dominating Set by at most one hop. Otherwise, we claim that $\mathrm{OPT}(I') \geq 2$. This is because if $\mathrm{OPT}(I') < 2$, then every vertex must be within distance 1, which implies the k-center that witnesses $\mathrm{OPT}(I')$ is a dominating set of I. Therefore, the $(2-\epsilon)$ approximation for k-center can be used to solve the Dominating Set Problem. This is impossible, unless $P = NP$. □

References

[1] A. Asadpour, M. Goemans, A. Madry, S. Oveis Gharan, and A. Saberi. An $O(\log n/\log\log n)$-approximation algorithm for the assymetric traveling salesman problem. *Proc. of ACM-SIAM SODA*, 2010.

[2] N. Christofides. Worst-case analysis of a new heuristic for the traveling salesman problem. *Technical Report, Graduate School of Industrial Administration, CMU* 1976.

[3] S. Sahni and T.F. Gonzalez. P-Complete Approximation Problems. *J. of the ACM* 23: 555-565, 1976.

[4] M. Held and R. M. Karp. The traveling salesman problem and minimum spanning trees. *Operations Research* 18: 1138–1162, 1970.

[5] W. L. Hsu and G.L. Nemhauser. Easy and hard bottleneck location problems. *Discrete Applied Mathematics*. 1:209-216, 1979.

[6] D. West. *Introductin to Graph Theory*. 2d ed. Prentice Hall. 277-278, 2001.

CS 598CSC: Approximation Algorithms	Lecture date: January 26, 2011
Instructor: Chandra Chekuri	Scribe: Md. Abul Hassan Samee (2009)

Notes edited by instructor in 2011.

1 Introduction

We discuss two closely related NP Optimization problems, namely Set Cover and Maximum Coverage in this lecture. Set Cover was among the first problems for which approximation algorithms were analyzed. This problem is also significant from a practical point of view, since the problem itself and several of its generalizations arise quite frequently in a number of application areas. We will consider three such generalizations of Set Cover in this lecture. We conclude the lecture with a brief discussion on how the Set Cover problem can be formulated in terms of submodular functions.

2 Set Cover and Maximum Coverage

2.1 Problem definition

In both the Set Cover and the Maximum Coverage problems, our input is a set $\mathcal{U}$ of n elements, and a collection $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ of m subsets of $\mathcal{U}$ such that $\bigcup_i S_i = \mathcal{U}$. Our goal in the Set Cover problem is to select as few subsets as possible from $\mathcal{S}$ such that their union covers $\mathcal{U}$. In the Maximum Coverage problem an integer $k \leq m$ is also specified in the input, and our goal is to select k subsets from $\mathcal{S}$ such that their union has the maximum cardinality. Note that the former is a minimization problem while the latter is a maximization problem. One can also consider weighted versions of these problems which we postpone to a later lecture.

2.2 Greedy approximation

Both Set Cover and Maximum Coverage are known to be **NP**-Hard. A natural greedy approximation algorithm for these problems is as follows.

```
Greedy Cover (U, S):
1: repeat
2:         pick the set that covers the maximum number of uncovered elements
3:         mark elements in the chosen set as covered
4: until done
```

In case of Set Cover, the algorithm Greedy Cover is *done* in line 4 when all the elements in set $\mathcal{U}$ have been covered. And in case of Maximum Coverage, the algorithm is *done* when exactly k subsets have been selected from $\mathcal{S}$.

2.3 Analysis of Greedy Cover

Theorem 1 Greedy Cover *is a* $1 - (1 - 1/k)^k \geq (1 - \frac{1}{e}) \simeq 0.632$ *approximation for* Maximum Coverage*, and a* $(\ln n + 1)$ *approximation for* Set Cover.

The following theorem due to Feige [1] implies that GREEDY COVER is essentially the best possible in terms of the approximation ratio that it guarantees in Theorem 1.

Theorem 2 *Unless* $\boldsymbol{NP} \subseteq \boldsymbol{DTIME}(n^{O(\log\log n)})$*, there is no* $(1-o(1))\ln n$ *approximation for* SET COVER*. Unless* $\boldsymbol{P}=\boldsymbol{NP}$*, for any fixed* $\epsilon > 0$*, there is no* $(1-\frac{1}{e}-\epsilon)$ *approximation for* MAXIMUM COVERAGE*.*

We proceed towards the proof of Theorem 1 by providing analysis of GREEDY COVER separately for SET COVER and MAXIMUM COVERAGE. Let OPT denote the value of an optimal solution to the MAXIMUM COVERAGE problem. Let x_i denote the number of *new* elements covered by GREEDY COVER in the i-th set that it picks. Also, let $y_i = \sum_{j=1}^{i} x_i$, and $z_i = OPT - y_i$. Note that, according to our notations, $y_0 = 0$, y_k is the number of elements chosen by GREEDY COVER, and $z_0 = OPT$.

Analysis for MAXIMUM COVERAGE

We have the following lemma for algorithm GREEDY COVER when applied on MAXIMUM COVERAGE.

Lemma 3 GREEDY COVER *is a* $1-(1-1/k)^k \geq 1-\frac{1}{e}$ *approximation for* MAXIMUM COVERAGE*.*

We first prove the following two claims.

Claim 4 $x_{i+1} \geq \frac{z_i}{k}$.

Proof: At each step, GREEDY COVER selects the subset S_j whose inclusion covers the maximum number of uncovered elements. Since the optimal solution uses k sets to cover OPT elements, some set must cover at least $1/k$ fraction of the at least z_i remaining uncovered elements from OPT. Hence, $x_{i+1} \geq \frac{z_i}{k}$. □

Claim 5 $z_{i+1} \leq (1-\frac{1}{k})^{i+1} \cdot OPT$

Proof: The claim is true for $i = 0$. We assume inductively that $z_i \leq (1-\frac{1}{k})^i \cdot OPT$. Then

$$\begin{aligned} z_{i+1} &\leq z_i - x_{i+1} \\ &\leq z_i(1-\frac{1}{k}) \quad \text{[using Claim 4]} \\ &\leq (1-\frac{1}{k})^{i+1} \cdot OPT. \end{aligned}$$

□

Proof of Lemma 3. It follows from Claim 5 that $z_k \leq (1-\frac{1}{k})^k \cdot OPT \leq \frac{OPT}{e}$. Hence, $y_k = OPT - z_k \geq (1-\frac{1}{e}) \cdot OPT$. □

Analysis for SET COVER

We have the following lemma.

Lemma 6 GREEDY COVER *is a* $(\ln n + 1)$ *approximation for* SET COVER*.*

Let k^* denote the value of an optimal solution to the SET COVER problem. Then an optimal solution to the MAXIMUM COVERAGE problem for $k = k^*$ would cover all the n elements in set $\mathcal{U}$, and $z_{k^*} \leq \frac{n}{e}$. Therefore, $\frac{n}{e}$ elements would remain uncovered after the first k^* steps of GREEDY COVER. Similarly, after $2 \cdot k^*$ steps of GREEDY COVER, $\frac{n}{e^2}$ elements would remain uncovered. This easy intuition convinces us that GREEDY COVER is a $(\ln n + 1)$ approximation for the SET COVER problem. A more succinct proof is given below.

Proof of Lemma 6. Since $z_i \leq (1 - \frac{1}{k^*})^i \cdot n$, after $t = k^* \ln \frac{n}{k^*}$ steps, $z_t \leq k^*$. Thus, after t steps, k^* elements are left to be covered. Since GREEDY COVER picks at least one element in each step, it covers all the elements after picking at most $k^* \ln \frac{n}{k^*} + k^* \leq k^*(\ln n + 1)$ sets. □

The following corollary readily follows from Lemma 6.

Corollary 7 *If $|S_i| \leq d$, then* GREEDY COVER *is a $(\ln d + 1)$ approximation for* SET COVER.

Proof: Since $k^* \geq \frac{n}{d}$, $\ln \frac{n}{k^*} \leq \ln d$. Then the claim follows from Lemma 6. □

Proof of Theorem 1. The claims follow directly from Lemma 3 and 6. □

A tight example for GREEDY COVER when applied on SET COVER

Let us consider a set $\mathcal{U}$ of n elements along with a collection $\mathcal{S}$ of $k+2$ subsets $\{R_1, R_2, C_1, C_2, \ldots, C_k\}$ of $\mathcal{U}$. Let us also assume that $|C_i| = 2^i$ and $|R_1 \cap C_i| = |R_2 \cap C_i| = 2^{i-1}$ $(1 \leq i \leq k)$, as illustrated in Fig. 1.

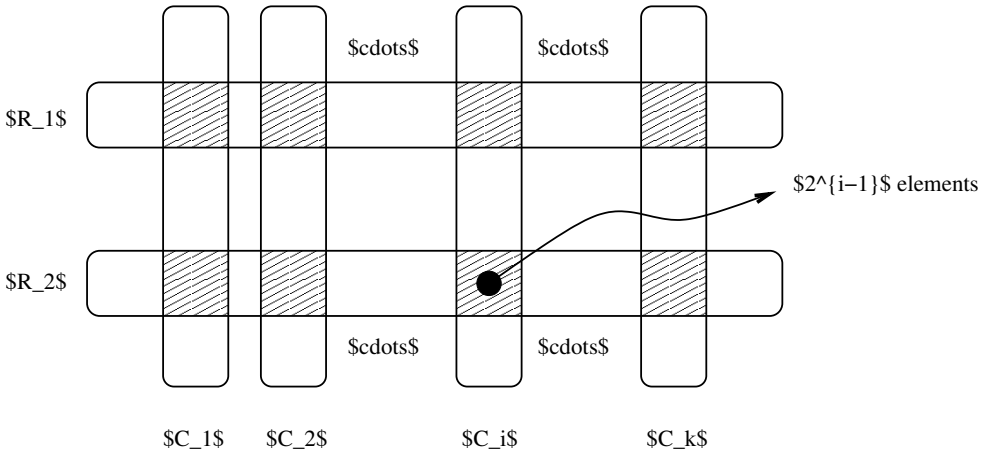


Figure 1: A tight example for GREEDY COVER when applied on SET COVER

Clearly, the optimal solution consists of only two sets, i.e., R_1 and R_2. Hence, $OPT = 2$. However, GREEDY COVER will pick each of the remaining k sets, namely $C_k, C_{k-1}, \ldots, C_1$. Since $n = 2 \cdot \sum_{i=0}^{k-1} 2^i = 2 \cdot (2^k - 1)$, we get $k \approx \Omega(\log_2 n)$. Hence the example is tight.

Exercise: Consider the weighted version of the SET COVER problem where a weight function $w : \mathcal{S} \to \mathcal{R}^+$ is given, and we want to select a collection $\mathcal{S}'$ of subsets from $\mathcal{S}$ such that $\cup_{X \in \mathcal{S}'} X = \mathcal{U}$, and $\sum_{X \in \mathcal{S}'} w(X)$ is the minimum. Prove that the greedy heuristic gives a $2 \cdot (\ln n + 1)$ approximation for this problem.

Hint 1: Note that the greedy algorithm never picks a set of cost more than OPT. *Hint 2:* By the first time the total cost of sets picked by the greedy algorithm exceeds OPT, it has covered a $(1 - 1/e)$ fraction of the elements.

3 Dominating Set and Vertex Cover

3.1 Dominating Set

A *dominating set* in a graph $G = (V, E)$ is a set $S \subseteq V$ such that for each $u \in V$, either $u \in S$, or some neighbor v of u is in S. In the Dominating Set problem, our goal is to find a smallest dominating set of G.

A natural greedy algorithm for this problem is to iteratively choose a vertex with the highest degree. It can be proved that this heuristic gives a $(\ln n + 1)$, or more accurately, a $(\ln(\Delta + 1) + 1)$ approximation for the Dominating Set problem.

Exercises:

1. Prove the approximation guarantees of the greedy heuristic for Dominating Set.
2. Show that Dominating Set is a special case of Set Cover.
3. Show that Set Cover can be reduced in an approximation preserving fashion to Dominating Set. More formally, show that if Dominating Set has an $\alpha(n)$-approximation where n is the number of vertices in the given instance then Set Cover has an $(1 - o(1))\alpha(n)$-approximation.

3.2 Vertex Cover

A *vertex cover* of a graph $G = (V, E)$ is a set $S \subseteq V$ such that for each edge $e \in E$, at least one end point of e is in S. In the Vertex Cover problem, our goal is to find a smallest vertex cover of G. In the *weighted* version of the problem, a weight function $w : V \to \mathcal{R}^+$ is given, and our goal is to find a minimum weight vertex cover of G. The unweighted version of the problem is also known as Cardinality Vertex Cover.

It can be shown that, the Greedy Cover algorithm can give an $O(\ln \Delta + 1)$ approximation for both weighted and unweighted versions of the Vertex Cover problem.

Exercises:

1. Show that Vertex Cover is a special case of Set Cover.
2. Construct an example that shows that Greedy Cover when applied on the Vertex Cover problem gives an $\Omega(\log n)$-approximation.

3.2.1 Better (constant) approximation for Vertex Cover

Cardinality Vertex Cover : The following is a 2-approximation algorithm for the Cardinality Vertex Cover problem.

```
Matching-VC (G):
1: S ← ∅
2: Compute a maximal matching M in G
3: for each edge (u, v) ∈ M do
4:         add both u and v to S
5: Output S
```

Theorem 8 Matching-VC *is a* 2*-approximation algorithm.*

The proof of Theorem 8 follows from two simple claims.

Claim 9 *Let OPT be the size of the vertex cover in an optimal solution. Then $OPT \geq |M|$.*

Proof: Since the optimal vertex cover must contain at least one end vertex of every edge in M, $OPT \geq |M|$. □

Claim 10 *Let $S(M) = \{u, v | (u, v) \in M\}$. Then $S(M)$ is a vertex cover.*

Proof: If $S(M)$ is not a vertex cover, then there must be an edge $e \in E$ such that neither of its endpoints are in M. But then e can be included in M, which contradicts the maximality of M. □

Proof of Theorem 8. Since $S(M)$ is a vertex cover, Claim 9 implies that $|S(M)| = 2 \cdot |M| \leq 2 \cdot OPT$. □

Weighted Vertex Cover: 2-approximation algorithms for the Weighted Vertex Cover problem can be designed based on LP rounding or Primal-Dual technique. These will be covered later in the course.

3.2.2 Set Cover with small frequencies

Vertex Cover is an instance of Set Cover where each element in $\mathcal{U}$ is in at most two sets (in fact, each element was in exactly two sets). This special case of the Set Cover problem has given us a 2-approximation algorithm. What would be the case if every element was contained in at most three sets? More generally, given an instance of Set Cover, for each $e \in \mathcal{U}$, let $f(e)$ denote the number of sets containing e. Let $f = \max_e f(e)$, which we call the *maximum frequency.*

Exercise: Give an f-approximation for Set Cover, where f isthe maximum frequency of an element. *Hint:* Follow the approach used for Vertex Cover .

4 Two important aspects of greedy approximation for Set Cover

4.1 Greedy approximation for implicit instances

It turns out that the universe $\mathcal{U}$ of elements and the collection $\mathcal{S}$ of subsets of $\mathcal{U}$ are not restricted to be finite or explicitly enumerated in the Set Cover problem. For instance, a problem could

require covering a finite set of points in the plane using disks of unit radius. There is an infinite set of such disks, but the greedy approximation algorithm can still be applied. For such implicit instances, the greedy algorithm can be used if we have access to an *oracle*, which, at each iteration, selects a set having the optimal density. However, an oracle may not always be capable of selecting an optimal set. In such cases, it may have to make the selections *approximately*. We call an oracle an α-*approximate oracle* if, at each iteration, it selects a set S such that $density(S) \geq \alpha \cdot Optimal\ Density$, for some $\alpha > 1$.

Exercise: Prove that the approximation guarantee of greedy approximation with an α-approximate oracle would be $\alpha(\ln n + 1)$ for Set Cover, and $(1 - \frac{1}{e^{\alpha}})$ for Maximum Coverage.

4.2 Greedy approximation for submodular functions

In a more general sense, the greedy approximation works for any *submodular set function*. Given a finite set E, a function $f : 2^E \to \mathcal{R}^+$ is submodular iff $f(A) + f(B) \geq f(A \cup B) + f(A \cap B)$ for all $A, B \subseteq E$. Alternatively, f is a submodular functions iff $f(A + i) - f(A) \geq f(B + i) - f(B)$ for all $i \in E$ and $A \subset B$. This second characterization is due to the property of *decreasing marginal utility* of submodular functions. Intuitively, adding element i to a set A will help at least as much as adding it to to a (larger) set $B \supset A$.

Exercise: Prove that the two characterizations of submodular functions are equivalent.

A submodular function $f(\cdot)$ is *monotone* if $f(A+i) \geq f(A)$ for all $i \in E$ and $A \subseteq E$. We assume that $f(\emptyset) = 0$. Submodular set functions arise in a large number of practical fields including combinatorial optimization, probability, and geometry. Examples include rank function of a matroid, the sizes of cutsets in a directed or undirected graph, the probability that a subset of events do not occur simultaneously, entropy of random variables, etc. In the following we show that the Set Cover and Maximum Coverage problems can be easily formulated in terms of submodular set functions.

Exercise. Suppose we are given a universe $\mathcal{U}$ and a collection $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ of subsets of $\mathcal{U}$. Now if we take $N = \{1, 2, \ldots, m\}$, $f : 2^N \to \mathcal{R}^+$, and define $f(A) = |\cup_{i \in A} S_i|$ for $A \subseteq E$, then show that the function f is submodular.

4.2.1 Submodular Set Cover

When formulated in terms of submodular set functions, the Set Cover problem is the following. Given a monotone submodular function f (whose value would be computed by an oracle) on $N = \{1, 2, \ldots, m\}$, find the smallest set $S \subseteq N$ such that $f(S) = f(N)$. Our previous greedy approximation can be applied to this formulation as follows.

```
Greedy Submodular (f, N):
1: S ← ∅
2: while f(S) ≠ f(N)
3:         find i to maximize f(S + i) − f(S)
4:         S ← S ∪ {i}
```

Exercises:

1. Prove that the greedy algorithm is a $1 + \ln(f(N))$ approximation for Submodular Set Cover.

2. Prove that the greedy algorithm is a $1 + \ln(\max_i f(i))$ approximation for Submodular Set Cover.

4.2.2 Submodular Maximum Coverage

By formulating the Maximum Coverage problem in terms of submodular functions, we seek to maximize $f(S)$ such that $|S| \leq k$. We can apply algorithm Greedy Submodular for this problem by changing the condition in line 2 to be: **while** $|S| \leq k$.

Note. For the Submodular Maximum Coverage problem, function f must be both submodular and monotone.
Exercise: Prove that greedy gives a $(1 - 1/e)$-approximation for Submodular Maximum Coverage problem.

References

[1] U. Feige. A Threshold of $\ln n$ for Approximating Set Cover. *J. of the ACM* 45(5): 634–652, 1998.

CS 598CSC: Approximation Algorithms	Lecture date: 1/28/11
Instructor: Chandra Chekuri	Scribe: Lewis Tseng (2009)

Notes edited by instructor in 2011.

We introduce the use linear programming (LP) in the design and analysis of approximation algorithms.The topics include Vertex Cover, Set Cover, randomized rounding, dual-fitting. It is assumed that the students have some background knowledge in basics of linear programming.

1 Vertex Cover via LP

Let $G = (V, E)$ be an undirected graph with arc weights $w : V \to R^+$. Recall the vertex cover problem from previous lecture. We can formulate it as an integer linear programming problem as follows. For each vertex v we have a variable x_v. We interpret the variable as follows: if $x_v = 1$ if v is chosen to be included in a vertex cover, otherwise $x_v = 0$. With this interprtation we can easily see that the minimum weight vertex cover can be formulated as the following integer linear program.

$$\begin{aligned} \min \quad & \sum_{v \in V} w_v x_v \\ \text{subject to} & \\ x_u + x_v \quad & \geq \quad 1 \qquad \forall e = (u, v) \in E \\ x_v \quad & \in \quad \{0, 1\} \qquad \forall v \in V \end{aligned}$$

However, solving integer linear programs is NP-Hard. Therefore we use Linear Programming (LP) to approximate the optimal solution, $\mathrm{OPT}(I)$, for the integer program. First, we can relax the constraint $x_v \in \{0, 1\}$ to $x_v \in [0, 1]$. It can be further simplified to $x_v \geq 0$, $\forall v \in V$.

Thus, a linear programming formulation for Vertex Cover is:

$$\begin{aligned} \min \quad & \sum_{v \in V} w_v x_v \\ \text{subject to} & \\ x_u + x_v \quad & \geq \quad 1 \qquad \forall e = (u, v) \in E \\ x_v \quad & \geq \quad 0 \end{aligned}$$

We now use the following algorithm:

Vertex Cover via LP:
Solve LP to obtain an optimal fractional solution x^*
Let $S = \{v \mid x^*_v \geq \frac{1}{2}\}$
Output S

Then the following claims are true:

Claim 1 *S is a vertex cover.*

Proof: Consider any edge, $e = (u, v)$. By feasibility of x^*, $x_u^* + x_v^* \geq 1$, and thus either $x_u^* \geq \frac{1}{2}$ or $x_v^* \geq \frac{1}{2}$. Therefore, at least one of u and v will be in S. □

Claim 2 $w(S) \leq 2\mathrm{OPT}_{LP}(I)$.

Proof: $\mathrm{OPT}_{LP}(I) = \sum_v w_v x_v^* \geq \frac{1}{2} \sum_{v \in S} w_v = \frac{1}{2} w(S)$ □

Therefore, $\mathrm{OPT}_{LP}(I) \geq \frac{\mathrm{OPT}(I)}{2}$ for all instances I.

Note: For minimization problems: $\mathrm{OPT}_{LP}(I) \leq \mathrm{OPT}(I)$, where $\mathrm{OPT}_{LP}(I)$ is the optimal solution found by LP; for maximization problems, $\mathrm{OPT}_{LP}(I) \geq \mathrm{OPT}(I)$.

Integrality Gap

We introduce the notion of *integrality gap* to show the best approximation guarantee we can acquire by using the LP optimum as a lower bound.

Definition: For a minimization problem Π, the integrality gap for a linear programming relaxation/formulation LP for Π is $\sup_{I \in \pi} \frac{\mathrm{OPT}(I)}{\mathrm{OPT}_{LP}(I)}$.

That is, the integrality gap is the worst case ratio, over all instances I of Π, of the integral optimal value and the fractional optimal value. Note that different linear programming formulations for the same problem may have different integrality gaps.

Claims 1 and 2 show that the integrality gap of the Vertex Cover LP formulation above is at most 2.

Question: Is this bound tight for the Vertex Cover LP?

Consider the following example: Take a complete graph, K_n, with n vertices, and each vertex has $w_v = 1$. It is clear that we have to choose $n - 1$ vertices to cover all the edges. Thus, $\mathrm{OPT}(K_n) = n - 1$. However, $x_v = \frac{1}{2}$ for each v is a feasible solution to the LP, which has a total weight of $\frac{n}{2}$. So gap is $2 - \frac{1}{n}$, which tends to 2 as $n \to \infty$.

Other Results on Vertex Cover

1. The current best approximation ratio for Vertex Cover is $2 - \Theta(\frac{1}{\sqrt{\log n}})$ [1].

2. Open problem: obtain a $2 - \varepsilon$ approximation or to prove that it is NP-hard to obtain $2 - \varepsilon$ for any fixed $\varepsilon > 0$. Current best hardness of approximation: unless P=NP, there is no 1.36 approximation for Vertex Cover [2].

3. The vertex cover problem can be solved optimally in polynomial time for bipartite graphs. This follows from what is known as K$\acute{}$onig's theorem.

4. The vertex cover problem admits a polynomial time approximation scheme (PTAS), that is a $(1 + \epsilon)$-approximation for any fixed $\epsilon > 0$, for planar graphs. This follows from a general approach due to Baker [**?**].

2 Set Cover via LP

The input to the Set Cover problem consists of a finite set $U = \{1, 2, ..., n\}$, and m subsets of U, $S_1, S_2, ..., S_n$. Each set S_j has a non-negative weigh w_j and the goal is to find the minimum weight collection of sets which cover all elements in U (in other words their union is U).

A linear programming relaxation for Set Cover is:

$$\begin{aligned} \min \quad & \sum_j w_j x_j \\ \text{subject to} & \\ \sum_{j: i \in S_j} x_j \quad & \geq \quad 1 \qquad \forall i \in \{1, 2, ..., n\} \\ x_j \quad & \geq \quad 0 \qquad 1 \leq j \leq m \end{aligned}$$

And its dual is:

$$\begin{aligned} \max \quad & \sum_{i=1}^{n} y_i \\ \text{subject to} & \\ \sum_{i \in S_j} y_i \quad & \leq \quad w_j \qquad \forall i \in \{1, 2, ..., n\} \\ y_i \quad & \geq \quad 0 \qquad \forall i \in 1, 2, ..., n \end{aligned}$$

We give several algorithms for Set Cover based on this primal/dual pair LPs.

2.1 Deterministic Rounding

Set Cover via LP:
Solve LP to obtain an optimal solution x^*, which contains fractional numbers.
Let $P = \{i \mid x_i^* \geq \frac{1}{f}\}$, where f is the maximum number of sets that contain any element
Output $\{S_j \mid j \in P\}$

Note that the above algorithm, even when specialized to vertex cover, is different from the one we saw earlier. It includes all sets which have a strictly positive value in an *optimum* solution the LP.

Let x^* be an optimal solution to the primal LP, y^* be an optimum solution to the dual, and let $P = \{j \mid x_j^* > 0\}$. First, note that by strong duality, $\sum_j w_j x_j^* = \sum_i y_i^*$. Second, by complementary slackness if $x_j^* > 0$ then the corresponding dual constraint is tight, that is $\sum_{i \in S_j} y_i^* = w_j$.

Claim 3 *The output of the algorithm is a feasible set cover for the given instance.*

Proof: Exercise. □

Claim 4 $\sum_{j \in P} w_j \leq f \sum_j w_j x_j^* = \text{OPT}_{LP}$.

Proof:

$$\sum_{j \in P} w_j = \sum_{j: x_j^* > 0} (w_j) = \sum_{j: x_j^* > 0} \left(\sum_{i \in S_j} y_i^* \right) = \sum_i y_i^* \left(\sum_{j: i \in S_j, x_j^* > 0} 1 \right) \leq f \sum_i y_i^* \leq f \mathrm{OPT}_{LP}(I).$$

. □

Notice that the the second equality is due to complementary slackness conditions (if $x_j > 0$, the corresponding dual constraint is tight), the penultimate inequality uses the definition of f, and the last inequality follows from weak duality (a feasible solution for the dual problem is a lower bound on the optimal primal solution).

Therefore we have that the algorithm outputs a cover of weight at most $f\mathrm{OPT}_{LP}$. We note that f can be as large as n in which case the bound given by the algorithm is quite weak. In fact, it is not construct examples that demonstrate the tightness of the analysis.

Remark: The analysis cruically uses the fact that x^* is an optimal solution. On the other hand the algorithm for vertex cover is more robust.

2.2 Randomized Rounding

Now we describe a different rounding that yields an approximation bound that does not depend on f.

Solving Set Cover via Randomized Rounding:
$A = \emptyset$, and let x^* be an optimal solution to the LP.
for $k = 1$ to $2 \ln n$ do
pick each S_j independently with probability x_j^*
if S_j is picked, $A = A \cup \{j\}$
end for
Output the sets with indices in A

Claim 5 *$Pr[i$ is not covered in an iteration$] = \prod_{j: i \in S_j} (1 - x_j^*) \leq \frac{1}{e}$.*

Intuition: We know that $\sum_{j: i \in S_j} x_j^* \geq 1$. Subject to this constraint, if and want to minimize the probability, we can let x_j^* equal to each other, then the probability $= (1 - \frac{1}{k})^k$, where $x_j^* = 1/k$.

Proof: $Pr[i$ is not covered in an iteration$] = \prod_{j: i \in S_j} (1 - x_j^*) \leq \prod_{j: i \in S_j} e^{-x_j^*} \leq e^{-\sum_{j: i \in S_j} x_j^*} \leq \frac{1}{e}$. □

We then obtain the following corollaries:

Corollary: $Pr[i$ is not covered at the end of the algorithm$] \leq e^{-2 \log n} \leq \frac{1}{n^2}$.

Corollary: $Pr[$all elements are covered, after the algorithm stops$] \geq 1 - \frac{1}{n}$. The above follows from the union bound. The probability that i is not covered is at most $1/n^2$, hence the probability that there is some i that is not covered is at most $n \cdot 1/n^2 \leq 1/n$.

Let C_t = cost of sets picked in iteration t, then $E[C_t] = \sum_{j=1}^{m} w_j x_j^*$, where $E[X]$ denotes the expectation of a random variable X. Then, let $C = \sum_{t=1}^{2\ln n} C_t$; we have $E[C] = \sum_{t=1}^{2\ln n} E[C_t] \leq 2\ln n \mathrm{OPT}_{LP}$. We know that $Pr[C > 2E[C]] \leq \frac{1}{2}$ by Markov's inequality, so we have $Pr[C \leq 4\ln n \mathrm{OPT}_{LP}] \geq \frac{1}{2}$. Therefore, $Pr[C \leq 4\ln n \mathrm{OPT}_{LP}$ and all items are covered$] \geq \frac{1}{2} - \frac{1}{n}$. Thus, the randomized rounding algorithm, with probability close to 1/2 succeeds in giving a feasible solution of cost $O(\log n)\mathrm{OPT}_{LP}$. Note that we can check whether the solution satisfies the desired properties (feasibility and cost) and repeat the algorithm if it does not.

1. We can check if solution after rounding satisfies the desired properties, such as all elements are covered, or cost at most $2c\log n \mathrm{OPT}_{LP}$. If not, repeat rounding. Expected number of iterations to succeed is a constant.

2. We can also use Chernoff bounds (large deviation bounds) to show that a single rounding succeeds with high probability (probability at least $1 - \frac{1}{poly(n)}$).

3. The algorithm can be *derandomized*. Derandomization is a technique of removing randomness or using as little randomness as possible. There are many derandomization techniques, such as the method of conditional expectation, discrepancy theory, and expander graphs. Broder *et al.* [5] use min-wise independent permutations to derandomize the RNC algorithm for approximate set cover due to S. Rajagopalan and V. Vazirani [6].

4. After a few rounds, select the cheapest set that covers each uncovered element. This has low expected cost. This algorithm ensures feasibility but guarantees cost only in the expected sense.

Other Results related to Set Cover

1. Unless P = NP, there is no $c\log n$ approximation for some fixed c [4].

2. Unless NP $\subseteq DTIME(n^{O(\log\log n)})$, there is no $(1 - o(1))\ln n$-approximation [3].

3. Unless P = NP, there is no $(1 - \frac{1}{e} + \varepsilon)$-approximation for max-coverage for any fixed $\varepsilon > 0$.

2.3 Dual-fitting

In this section, we introduce the technique of dual-fitting for the analysis of approximation algorithms. At a high-level the approach is the following:

1. Construct a feasible solution to the dual LP.

2. Show that the cost of the solution returned by the algorithm can be bounded in terms of the value of the dual solution.

Note that the algorithm itself need not be LP based. Here, we use Set Cover as an example. Please refer to the previous section for the primal and dual LP formulations of Set Cover.

We can interpret the dual as follows: Think of y_i as how much element i is willing to pay to be covered; the dual maximizes the total payment, subject to the constraint that for each set, the total payment of elements in that set is at most the cost of the set.

The greedy algorithm for weighted Set Cover is as follows:

GREEDY SET COVER:
$Covered = \emptyset$;
$A = \emptyset$;
While $Covered \neq U$ do
  $j \leftarrow \arg\min_k(\frac{w_k}{|S_k \cap Uncovered|})$;
  $Covered = Covered \cup S_j$;
  $A = A \cup \{j\}$.
end while;
Output sets in A as cover

Theorem 6 GREEDY SET COVER *picks a solution of cost* $\leq H_d \cdot \mathrm{OPT}_{LP}$*, where* d *is the maximum set size, i.e.,* $d = \max_j |S_j|$.

To prove this, we can augment the algorithm a little bit:

AUGMENTED GREEDY ALGORITHM OF WEIGHTED SET COVER:
$Covered = \emptyset$;
while $Covered \neq U$ do
  $j \leftarrow \arg\min_k(\frac{w_k}{|S_k \cap Uncovered|})$;
  if i is uncovered and $i \in S_j$, set $p_i = \frac{w_j}{|S_j \cap Uncovered|}$;
  $Covered = Covered \cup S_j$;
  $A = A \cup \{j\}$.
end while;
Output sets in A as cover

It is easy to see that the algorithm outputs a set cover.

Claim 7 $\sum_{j \in A} w_j = \sum_i p_i$.

Proof: Consider when j is added to A. Let $S'_j \subseteq S_j$ be the elements that are uncovered before j is added. For each $i \in S'_j$ the algorithm sets $p_i = w_j/|S'_j|$. Hence, $\sum_{i \in S'_j} p_i = w_j$. Moreover, it is easy to see that the sets S'_j, $j \in A$ are disjoint and together partition U. Therefore,

$$\sum_{j \in A} w_j = \sum_{j \in A} \sum_{i \in S'_j} p_i = \sum_{i \in U} p_i.$$

□

For each i, let $y'_i = \frac{1}{H_d} p_i$.

Claim 8 *y' is a feasible solution for the dual LP.*

Suppose the claim is true, then the cost of GREEDY SET COVER's solution $= \sum_i p_i = H_d \sum_i y'_i \leq H_d \mathrm{OPT}_{LP}$. The last step is because any feasible solution for the dual problem is a lower bound on the value of the primal LP (weak duality).

Now, we prove the claim. Let S_j be an arbitrary set, and let $|S_j| = t \leq d$. Let $S_j = \{i_1, i_2, ..., i_t\}$, where we the elements are ordered such that i_1 is covered by Greedy no-later than i_2, and i_2 is covered no later than i_3 and so on.

Claim 9 *For* $1 \leq h \leq t$, $p_{i_h} \leq \frac{w_j}{t-h+1}$.

Proof: Let $S_{j'}$ be the set that covers i_h in Greedy. When Greedy picked $S_{j'}$ the elements $i_h, i_{h+1}, \ldots, i_t$ from S_j were uncovered and hence Greedy could have picked S_j as well. This implies that the density of $S_{j'}$ when it was picked was no more than $\frac{w_j}{t-h+1}$. Therefore p_{i_h} which is set to the density of $S_{j'}$ is at most $\frac{w_j}{t-h+1}$. □

From the above claim, we have

$$\sum_{1 \leq h \leq t} p_{i_h} \leq \sum_{1 \leq h \leq t} \frac{w_j}{t-h+1} = w_j H_t \leq w_j H_d.$$

Thus, the setting of y'_i to be p_i scaled down by a factor of H_d gives a feasible solution.

References

[1] G. Karakostas. A better approximation ratio for the Vertex Cover problem. *ECCC Report* TR04-084, 2004.

[2] I. Dinur and S. Safra. The importance of being biased. *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, pages 33-42, 2002.

[3] U. Feige. A Threshold of $\ln n$ for Approximating Set Cover. *Journal of the ACM* , v.45 n.4, p.634 - 652, 1998.

[4] R. Raz and M. Safra. A sub-constant error-probability low-degree test, and a sub-constant error-probability PCP characterization of NP. *Proceedings of STOC 1997*, pp. 475-484, 1997.

[5] A. Z. Broder, M. Charikar, and M. Mitzenmacher. A derandomization using min-wise independent permutations *Journal of Discrete Algorithms*, Volume 1, Issue 1, pages 11-20, 2003.

[6] S. Rajagopalan and V. Vazirani. Primal-Dual RNC Approximation Algorithms for Set Cover and Covering Integer Programs *SIAM Journal on Computing*, Volume 28, Issue 2, p.525 - 540, 1999.

CS 598CSC: Approximation Algorithms	Lecture date: February 9, 2011
Instructor: Chandra Chekuri	Scribe: Kyle Fox (2009)

In this lecture we explore the KNAPSACK problem. This problem provides a good basis for learning some important procedures used for approximation algorithms that give better solutions at the cost of higher running time.

1 The Knapsack Problem

1.1 Problem Description

In the KNAPSACK problem we are given a knapsack capacity B, and set N of n items. Each item i has a given size $s_i \geq 0$ and a profit $p_i \geq 0$. Given a subset of the items $A \subseteq N$, we define two functions, $s(A) = \sum_{i \in A} s_i$ and $p(A) = \sum_{i \in A} p_i$, representing the total size and profit of the group of items respectively. The goal is to choose a subset of the items, A, such that $s(A) \leq B$ and $p(A)$ is maximized. We will assume every item has size at most B. It is not difficult to see that if all the profits are 1, the natural greedy algorithm of sorting the items by size and then taking the smallest items will give an optimal solution. Assuming the profits and sizes are integral, we can still find an optimal solution to the problem relatively quickly using dynamic programming in either $O(nB)$ or $O(nP)$ time, where $P = \sum_{i=1}^{n} p_i$. Finding the details of these algorithms was given as an exercise in the practice homework. While these algorithms appear to run in polynomial time, it should be noted that B and P can be exponential in the size of the input assuming the numbers in the input are not written in unary. We call these *pseudo-polynomial time algorithms* as their running times are polynomial only when numbers in the input are given in unary.

1.2 A Greedy Algorithm

Consider the following greedy algorithm for the KNAPSACK problem which we will refer to as GREEDYKNAPSACK. We sort all the items by the ratio of their profits to their sizes so that $\frac{p_1}{s_1} \geq \frac{p_2}{s_2} \geq \cdots \geq \frac{p_n}{s_n}$. Afterward, we greedily take items in this order as long as adding an item to our collection does not exceed the capacity of the knapsack. It turns out that this algorithm can be arbitrarily bad. Suppose we only have two items in N. Let $s_1 = 1$, $p_1 = 2$, $s_2 = B$, and $p_2 = B$. GREEDYKNAPSACK will take only item 1, but taking only item 2 would be a better solution. As it turns out, we can easily modify this algorithm to provide a 2-approximation by simply taking the best of GREEDYKNAPSACK's solution or the most profitable item. We will call this new algorithm MODIFIEDGREEDY.

Theorem 1 MODIFIEDGREEDY *has an approximation ratio of* $1/2$ *for the* KNAPSACK *problem.*

Proof: Let k be the index of the first item that is not accepted by GREEDYKNAPSACK. Consider the following claim:

Claim 2 $p_1 + p_2 + \ldots p_k \geq \text{OPT}$. *In fact,* $p_1 + p_2 + \cdots + \alpha p_k \geq \text{OPT}$ *where* $\alpha = \frac{B - (s_1 + s_2 + \cdots + s_{k-1})}{s_k}$ *is the fraction of item* k *that can still fit in the knapsack after packing the first* $k-1$ *items.*

The proof of Theorem 1 follows immediately from the claim. In particular, either $p_1 + p_2 + \cdots + p_{k-1}$ or p_k must be at least OPT/2. We now only have to prove Claim 2. We give an LP relaxation of the KNAPSACK problem as follows: Here, $x_i \in [0, 1]$ denotes the fraction of item i packed in the knapsack.

$$\begin{aligned} \text{maximize } & \sum_{i=1}^{n} p_i x_i \\ \text{subject to } & \sum_{i=1}^{n} s_i x_i \leq B \\ & x_i \leq 1 \text{ for all } i \text{ in } \{1 \ldots n\} \\ & x_i \geq 0 \text{ for all } i \text{ in } \{1 \ldots n\} \end{aligned}$$

Let OPT$'$ be the optimal value of the objective function in this linear programming instance. Any solution to KNAPSACK is a feasible solution to the LP and both problems share the same objective function, so OPT$' \geq$ OPT. Now set $x_1 = x_2 = \cdots = x_{k-1} = 1$, $x_k = \alpha$, and $x_i = 0$ for all $i > k$. This is a feasible solution to the LP that cannot be improved by changing any one tight constraint, as we sorted the items. Therefore, $p_1 + p_2 + \cdots + \alpha p_k = \text{OPT}' \geq \text{OPT}$. The first statement of the lemma follows from the second as $\alpha \leq 1$. □

1.3 A Polynomial Time Approximation Scheme

Using the results from the last section, we make a few simple observations. Some of these lead to a better approximation.

Observation 3 *If for all i, $s_i \leq \epsilon B$,* GREEDYKNAPSACK *gives a $(1 - \epsilon)$ approximation.*

This follows from the deductions below:

$$\begin{aligned} \text{For } 1 \leq i \leq k,\ p_i/s_i &\geq p_k/s_k \\ \Rightarrow p_1 + p_2 + \cdots + p_k &\geq (s_1 + s_2 + \cdots + s_k) p_k/s_k \\ \Rightarrow p_k &\leq s_k(p_1 + p_2 + \cdots + p_k)/B \\ &\leq \epsilon(p_1 + p_2 + \cdots + p_k) \\ &\leq \epsilon(p_1 + p_2 + \cdots + p_{k-1})/(1 - \epsilon) \\ \Rightarrow p_1 + p_2 + \cdots + p_{k-1} &\geq (1 - \epsilon)\text{OPT} \end{aligned}$$

The third line follows because $s_1 + s_2 + \cdots + s_k > B$, and the last from Claim 2.

Observation 4 *If for all i, $p_i \leq \epsilon$*OPT*,* GREEDYKNAPSACK *gives a $(1 - \epsilon)$ approximation.*

This follows immediately from Claim 2.

Observation 5 *There are at most $\lceil \frac{1}{\epsilon} \rceil$ items with profit at least ϵ*OPT *in any optimal solution.*

We may now describe the following algorithm. Let $\epsilon \in (0, 1)$ be a fixed constant and let $h = \lceil \frac{1}{\epsilon} \rceil$. We will try to guess the h most profitable items in an optimal solution and pack the rest greedily.

GUESS H + GREEDY(N, B):
For each $S \subseteq N$ such that $|S| \leq h$:
  Pack S in knapsack of size at most B
  Let i be the least profitable item in S. Remove all items $j \in N - S$ where $p_j > p_i$.
  Run GREEDYKNAPSACK on remaining items with remaining capacity $B - \sum_{i \in S} s_i$
Output best solution from above

Theorem 6 GUESS H + GREEDY *gives a* $(1 - \epsilon)$ *approximation and runs in* $O(n^{\lceil 1/\epsilon \rceil + 1})$ *time.*

Proof: For the running time, observe that there are $O(n^h)$ subsets of N. For each subset, we spend linear time greedily packing the remaining items. The time initially spent sorting the items can be ignored thanks to the rest of the running time.

For the approximation ratio, consider a run of the loop where S actually is the h most profitable items in an optimal solution and the algorithm's greedy stage packs the set of items $A' \subseteq (N - S)$. Let OPT' be the optimal way to pack the smaller items in $N - S$ so that $\text{OPT} = p(S) + \text{OPT}'$. Let item k be the first item rejected by the greedy packing of $N - S$. We know $p_k \leq \epsilon\text{OPT}$ so by Claim 2 $p(A') \geq \text{OPT}' - \epsilon\text{OPT}$. This means the total profit found in that run of the loop is $p(S) + p(A') \geq (1 - \epsilon)\text{OPT}$. □

Note that for any fixed choice of epsilon, the algorithm above runs in polynomial time. This type of algorithm is known as a *polynomial time approximation scheme* or PTAS. We say a maximization problem Π has a PTAS if for all $\epsilon > 0$, there exists a polynomial time algorithm that gives a $(1 - \epsilon)$ approximation ($(1 + \epsilon)$ for minimization problems). In general, one can often find a PTAS for a problem by greedily filling in a solution after first searching for a good basis on which to work. As described below, KNAPSACK actually has something stronger known as a *fully polynomial time approximation scheme* or FPTAS. A maximization problem Π has a FPTAS if for all $\epsilon > 0$, there exists an algorithm that gives a $(1-\epsilon)$ approximation ($(1+\epsilon)$ for minimization problems) and runs in time polynomial in both the input size and $1/\epsilon$.

1.4 Rounding and Scaling

Earlier we mentioned exact algorithms based on dynamic programming that run in $O(nB)$ and $O(nP)$ time but noted that B and P may be prohibitively large. If we could somehow decrease one of those to be polynomial in n without losing too much information, we might be able to find an approximation based on one of these algorithms. Let $p_{\max} = \max_i p_i$ and note the following.

Observation 7 $p_{\max} \leq \text{OPT} \leq np_{\max}$

Now, fix some $\epsilon \in (0, 1)$. We want to scale the profits and round them to be integers so we may use the $O(nP)$ algorithm efficiently while still keeping enough information in the numbers to allow for an accurate approximation. For each i, let $p'_i = \lfloor \frac{n}{\epsilon} \frac{1}{p_{\max}} p_i \rfloor$. Observe that $p'_i \leq \frac{n}{\epsilon}$ so now the sum of the profits P' is at most $\frac{n^2}{\epsilon}$. Also, note that we lost at most n profit from the scaled optimal solution during the rounding, but the scaled down OPT is still at least $\frac{n}{\epsilon}$. We have only lost an ϵ fraction of the solution. This process of rounding and scaling values for use in exact algorithms has use in a large number of other maximization problems. We now formally state the algorithm ROUND&SCALE and prove its correctness and running time.

ROUND&SCALE(N, B):
For each i set $p'_i = \lfloor \frac{n}{\epsilon} \frac{1}{p_{\max}} p_i \rfloor$
Run exact algorithm with run time $O(nP')$ to obtain A
Output A

Theorem 8 ROUND&SCALE *gives a* $(1-\epsilon)$ *approximation and runs in* $O(\frac{n^3}{\epsilon})$ *time.*

Proof: The rounding can be done in linear time and as $P' = O(\frac{n^2}{\epsilon})$, the dynamic programing portion of the algorithm runs in $O(\frac{n^3}{\epsilon})$ time. To show the approximation ratio, let $\alpha = \frac{n}{\epsilon} \frac{1}{p_{\max}}$ and let A be the solution returned by the algorithm and A^* be the optimal solution. Observe that for all $X \subseteq N$, $\alpha p(X) - |X| \leq p'(X) \leq \alpha p(X)$ as the rounding lowers each scaled profit by at most 1. The algorithm returns the best choice for A given the scaled and rounded values, so we know $p'(A) \geq p'(A^*)$.

$$p(A) \geq \frac{1}{\alpha} p'(A) \geq \frac{1}{\alpha} p'(A^*) \geq p(A^*) - \frac{n}{\alpha} = \text{OPT} - \epsilon p_{\max} \geq (1-\epsilon)\text{OPT}$$

□

It should be noted that this is not the best FPTAS known for KNAPSACK. In particular, [1] shows a FPTAS that runs in $O(n \log(1/\epsilon) + 1/\epsilon^4)$ time.

2 Other Problems

The following problems are related to Knapsack; they can also be viewed as special cases of the set covering problems discussed previously. Can you see how?

2.1 Bin Packing

BINPACKING gives us n items as in KNAPSACK. Each item i is given a size $s_i \in (0, 1]$. The goal is to find the minimum number of bins of size 1 that are required to pack all of the items. As a special case of SET COVER, there is an $O(\log n)$ approximation algorithm for BINPACKING, but it is easy to see that one can do much better. In fact most greedy algorithms give a factor of 2 or better. There is an algorithm for BINPACKING that guarantees a packing of size $(1+\epsilon)\text{OPT} + 1$.

2.2 Multiple Knapsack

MULTIPLEKNAPSACK gives us m knapsacks of the same size B and n items with sizes and profits as in KNAPSACK. We again wish to pack items to obtains as large a profit as possible, except now we have more than one knapsack with which to do so. The obvious greedy algorithm based on MAXIMUM COVERAGE yields a $(1 - 1/e)$ approximation; can you do better?

References

[1] E. L. Lawler. Fast Approximation Algorithms for Knapsack Problems. *Mathematics of Operations Research*, 4(4): 339–356, 1979.

CS 598CSC: Approximation Algorithms	Lecture date: February 11, 2011
Instructor: Chandra Chekuri	Scribe: CC

In the previous lecture we discussed the KNAPSACK problem. In this lecture we discuss other *packing* and *independent set* problems.

1 Maximum Independent Set Problem

A basic graph optimization problem with many applications is the *maximum (weighted) independent set problem* (MIS) in graphs.

Definition 1 *Given an undirected graph $G = (V, E)$ a subset of nodes $S \subseteq V$ is an independent set (stable set) iff there is no edge in E between any two nodes in S. A subset of nodes S is a* clique *if every pair of nodes in S have an edge between them in G.*

The MIS problem is the following: given a graph $G = (V, E)$ find an independent set in G of maximum cardinality. In the weighted case, each node $v \in V$ has an associated non-negative weight $w(v)$ and the goal is to find a maximum weight independent set. This problem is NP-Hard and it is natural to ask for approximation algorithms. Unfortunately, as the famous theorem below shows, the problem is extremely hard to approximate.

Theorem 1 (Håstad [1]) *Unless $P = NP$ there is no $\frac{1}{n^{1-\epsilon}}$-approximation for MIS for any fixed $\epsilon > 0$ where n is the number of nodes in the given graph.*

Remark: The maximum clique problem is to find the maximum cardinality clique in a given graph. It is approximation-equivalent to the MIS problem; simple complement the graph.

The theorem basically says the following: there are a class of graphs in which the maximum independent set size is either less than n^{δ} or greater than $n^{1-\delta}$ and it is NP-Complete to decide whether a given graph falls into the former category or the latter.

The lower bound result suggests that one should focus on special cases, and several interesting positive results are known. First, we consider a simple greedy algorithm for the unweighted problem.

```
GREEDY(G):
S ← ∅
While G is not empty do
   Let v be a node of minimum degree in G
   S ← S ∪ {v}
   Remove v and its neighbors from G
end while
Output S
```

Theorem 2 *Greedy outputs an independent set S such that $|S| \geq n/(\Delta + 1)$ where Δ is the maximum degree of any node in the graph.*

Proof: We upper bound the number of nodes in $V \setminus S$ as follows. A node u is in $V \setminus S$ because it is removed as a neighbor of some node $v \in S$ when Greedy added v to S. Charge u to v. A node $v \in S$ can be charged at most Δ times since it has at most Δ neighbors. Hence we have that $|V \setminus S| \leq \Delta|S|$. Since every node is either in S or $V \setminus S$ we have $|S| + |V \setminus S| = n$ and therefore $(\Delta + 1)|S| \geq n$ which implies that $|S| \geq n/(\Delta + 1)$. □

Since the maximum independent set size in a graph is n we obtain the following.

Corollary 3 *Greedy gives a $\frac{1}{\Delta+1}$-approximation for (unweighted)* MIS *in graphs of degree at most Δ.*

Exercise: Show that Greedy outputs an independent set of size at least $\frac{n}{2(d+1)}$ where d is the *average* degree of G.

Remark: The well-known Turan's theorem shows via a clever argument that there is always an independent set of size $\frac{n}{(d+1)}$ where d is the average degree of G.

Remark: For the case of unweighted graphs one can obtain an approximation ratio of $\Omega(\frac{\log d}{d \log \log d})$ where d is the average degree. Surprisingly, under a complexity theory conjecture called the Unique-Games conjecture it is known to be NP-Hard to approximate MIS to within a factor of $O(\frac{\log^2 \Delta}{\Delta})$ in graphs with maximum degree Δ when Δ is sufficiently large.

Exercise: Consider the weigthed MIS problem on graphs of maximum degree Δ. Alter Greedy to sort the nodes in non-increasing order of the weight and show that it gives a $\frac{1}{\Delta+1}$-approximation. Can one obtain an $\Omega(1/d)$-approximation for the weighted case where d is the average degree?

LP Relaxation: One can formulate a simple linear-programming relaxation for the (weighted) MIS problem where we have a variable $x(v)$ for each node $v \in V$ indicating whether v is chosen in the independent set or not. We have constraints which state that for each edge (u, v) only one of u or v can be chosen.

$$\begin{aligned} \text{maximize } & \sum_{v \in V} w(v)x(v) \\ \text{subject to } & x(u) + x(v) \leq 1 \qquad (u, v) \in E \\ & x(v) \in [0, 1] \qquad v \in V \end{aligned}$$

Although the above is a valid integer programming relaxation of MIS when the variabels are constrained to be in $\{0, 1\}$, it is not a particularly useful formulation for the following simple reason.

Claim 4 *For* any *graph the optimum value of the above LP relaxation is at least $w(V)/2$. In particular, for the unweighted case it is at least $n/2$.*

Simply set each $x(v)$ to 1/2!

One can obtain a *strengthened* formulation below by observing that if S is clique in G then any independent set can pick at most one node from S.

$$\begin{aligned} \text{maximize} \sum_{v \in V} w(v)x(v) & \\ \text{subject to} \sum_{v \in S} x(v) \leq 1 & \qquad S \text{ is a clique in } G \\ x(v) \in [0,1] & \qquad v \in V \end{aligned}$$

The above linear program has an exponential number of variables and it cannot be solved in polynomial time in general but for some special cases of interest the above linear program can indeed be solved (or approximately solved) in polynomial time and leads to either exact algorithms or good approximation bounds.

Approximability of VERTEX COVER **and** MIS**:** The following is a basic fact and is easy to prove.

Proposition 5 *In any graph $G = (V, E)$, S is a vertex cover in G if and only if $V \setminus S$ is an independent set in G. Thus $\alpha(G) + \beta(G) = |V|$ where $\alpha(G)$ is the size of a maximum independent set in G and $\beta(G)$ is the size of a minimum vertex cover in G.*

The above shows that if one of VERTEX COVER or MIS is NP-Hard then the other is as well. We have seen that VERTEX COVER admits a 2-approximation while MIS admits no constant factor approximation. It is useful to see why a 2-approximation for VERTEX COVER does not give any useful information for MIS even though $\alpha(G) + \beta(G) = |V|$. Suppose S^* is an optimal vertex cover and has size $\geq |V|/2$. Then a 2-approximation algorithm is only guaranteed to give a vertex cover of size $|V|$! Hence one does not obtain a non-trivial independent set by complementing the approximate vertex cover.

Some special cases of MIS**:** We mention some special cases of MIS that have been considered in the literature, this is by no means an exhaustive list.

- Interval graphs; these are intersection graphs of intervals on a line. An exact algorithm can be obtained via dynamic programming and one can solve more general versions via linear programming methods.
- Note that a maximum (weight) matching in a graph G can be viewed as a maximum (weight) independent set in the line-graph of G and can be solved exactly in polynomial time. This has been extended to what are known as claw-free graphs.
- Planar graphs and generalizations to bounded-genus graphs, and graphs that exclude a fixed minor. For such graphs one can obtain a PTAS due to ideas originally from Brenda Baker.
- Geometric intersection graphs. For example, given n disks on the plane find a maximum number of disks that do not overlap. One could consider other (convex) shapes such as axis parallel rectangles, line segments, pseudo-disks etc. A number of results are known. For example a PTAS is known for disks in the plane. An $\Omega(\frac{1}{\log n})$-approximation for axis-parallel rectangles in the plane when the rectangles are weighted and an $\Omega(\frac{1}{\log \log n})$-approximation for the unweighted case.

2 Packing Integer Programs (PIPs)

We can express the KNAPSACK problem as the following integer program. We scaled the knapsack capacity to 1 without loss of generality.

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{n} p_i x_i \\ \text{subject to} \quad & \sum_{i} s_i x_i \leq 1 \\ & x_i \in \{0,1\} \qquad 1 \leq i \leq n \end{aligned}$$

More generally if have multiple linear constraints on the "items" we obtain the following integer program.

Definition 2 *A packing integer program (PIP) is an integer program of the form* $\max\{wx \mid Ax \leq 1, x \in \{0,1\}^n\}$ *where* w *is a* $1 \times n$ *non-negative vector and* A *is a* $m \times n$ *matrix with entries in* $[0,1]$. *We call it a* $\{0,1\}$*-PIP if all entries are in* $\{0,1\}$.

In some cases it is useful/natural to define the problem as $\max\{wx \mid Ax \leq b, x \in \{0,1\}^n\}$ where entries in A and b are required to rational/integer valued. We can convert it into the above form by dividing each row of A by b_i.

When m the number of rows of A (equivalently the constraints) is small the problem is tractable. It is some times called the m-dimensional knapsack (recall the problem in HW 1) and one can obtain a PTAS for any fixed constant m. However, when m is large we observe that MIS can be cast as a special case of $\{0,1\}$-PIP. It corresponds exactly to the simple integer/linear program that we saw in the previous section. Therefore the problem is at least as hard to approximate as MIS. Here we show via a clever LP-rounding idea that one can generalize the notion of bounded-degree to *column-sparsity* in PIPs and obtain a related approximation. We will then introduce the notion of *width* of the constraints and show how it allows for improved bounds.

Definition 3 *A PIP is* k*-column-sparse if the number of non-zero entries in each column of* A *is at most* k. *A PIP has width* W *if* $\max_{i,j} A_{ij}/b_i \leq 1/W$.

2.1 Randomized Rounding with Alteration for PIPs

We saw that randomized rounding gave an $O(\log n)$ approximation algorithm for the SET COVER problem which is a canonical covering problem. Here we will consider the use of randomized rounding for packing problems. Let x be an optimum fractional solution to the natural LP relaxation of a PIP where we replace the constraint $x \in \{0,1\}^n$ by $x \in [0,1]^n$. Suppose we apply independent randomized rounding where we set x'_i to 1 with probability x_i. Let x' be the resulting integer solution. The expected weight of this solution is exactly $\sum_i w_i x_i$ which is the LP solution value. However, x' may not satisfy the constraints given by $Ax \leq b$. A natural strategy to try to satisfy the constraints is to set x'_1 to 1 with probability cx_i where $c < 1$ is some scaling constant. This may help in satisfying the constraints because the scaling creates some room in the constraints; we now have that the expected solution value is $c \sum_i w_i x_i$, a loss of a factor of c. Scaling by itself does not

allow us to claim that all constraints are satisfied with good probability. A very useful technique in this context is the technique of *alteration*; we judiciously fix/alter the rounded solution x' to force it to satisfy the constraints by setting some of the variables that are 1 in x' to 0. The trick is to do this in such a way as to have a handle on the final probability that a variable is set to 1. We will illustrate this for the KNAPSACK problem and then generalize the idea to k-sparse PIPs. The algorithms we present are from [2].

Rounding for Knapsack: Consider the KNAPSACK problem. It is convenient to think of this in the context of PIPs. So we have $ax \leq 1$ where a_i now represents the size of item i and the knapsack capacity is 1; w_i is the weight of item. Suppose x is a fractional solution. Call an item i "big" if $a_i > 1/2$ and otherwise it is "small". Let S be the indices of small items and B the indices of the big items. Consider the following rounding algorithm.

ROUNDING-WITH-ALTERATION FOR KNAPSACK:
Let x be an optimum fractional solution
Round each i to 1 independently with probability $x_i/4$. Let x' be rounded solution.
$x'' = x'$
If ($x'_i = 1$ for exactly one big item i)
  For each $j \neq i$ set $x''_j = 0$
Else If ($\sum_{i \in S} s_i x'_i > 1$ *or* two or more big items are chosen in x')
  For each j set $x''_j = 0$
End If
Output feasible solution x''

In words, the algorithm alters the rounded solution x' as follows. If exactly one big item is chosen in x' then the algorithm retains that item and rejects all the other small items. Otherwise, the algorithm rejects all items if two or more big items are chosen in x' or if the total size of all small items chosen in x' exceeds the capacity.

The following claim is easy to verify.

Claim 6 *The integer solution x'' is feasible.*

Now let us analyze the probability of an item i being present in the final solution. Let $\mathcal{E}_1$ be the event that $\sum_{i \in S} a_i x'_i > 1$, that is the sum of the sizes of the small items chose in x' exceeds the capacity. Let $\mathcal{E}_2$ be the event that at least one big item is chosen in x'.

Claim 7 $\Pr[\mathcal{E}_1] \leq 1/4$.

Proof: Let $X_s = \sum_{i \in S} a_i x'_i$ be the random variable that measures the sum of the sizes of the small items chosen. We have, by linearity of expectation, that

$$\mathbb{E}[X_s] = \sum_{i \in S} a_i \mathbb{E}[x'_i] = \sum_{i \in S} a_i x_i/4 \leq 1/4.$$

By Markov's inequality, $\Pr[X_s > 1] \leq \mathbb{E}[X_s]/1 \leq 1/4$. □

Claim 8 $\Pr[\mathcal{E}_2] \leq 1/2$.

Proof: Since the size of each big item in B is at least $1/2$, we have $1 \geq \sum_{i \in B} a_i x_i \geq \sum_{i \in B} x_i/2$. Therefore $\sum_{i \in B} x_i/4 \leq 1/2$. Event $\mathcal{E}_2$ happens if some item $i \in B$ is chosen in the random selection. Since i is chosen with probability $x_i/4$, by the union bound, $\Pr[\mathcal{E}_2] \leq \sum_{i \in B} x_i/4 \leq 1/2$. □

Lemma 9 *Let Z_i be the indicator random variable that is 1 if $x''_i = 1$ and 0 otherwise. Then $\mathbb{E}[Z_i] = \Pr[Z_i = 1] \geq x_i/16$.*

Proof: We consider the binary random variable X_i which is 1 if $x'_i = 1$. We have $\mathbb{E}[X_i] = \Pr[X_i = 1] = x_i/4$. We write

$$\Pr[Z_i = 1] = \Pr[X_i = 1] \cdot \Pr[Z_i = 1 \mid X_i = 1] = \frac{x_i}{4} \Pr[Z_i = 1 \mid X_i = 1].$$

To lower bound $\Pr[Z_i = 1 \mid X_i = 1]$ we upper bound the probability $\Pr[Z_i = 0 | X_i = 1]$, that is, the probability that we reject i conditioned on the fact that it is chosen in the random solution x'.

First consider a big item i that is chosen in x'. Then i is rejected iff if *another* big item is chosen in x'; the probability of this can be upper bounded by $\Pr[\mathcal{E}_1]$. If item i is small then it is rejected if and only if $\mathcal{E}_2$ happens or if a big item is chosen which happens with $\Pr[\mathcal{E}_1]$. In either case

$$\Pr[Z_i = 0 | X_i = 1] \leq \Pr[\mathcal{E}_1] + \Pr[\mathcal{E}_2] \leq 1/4 + 1/2 = 3/4.$$

Thus,

$$\Pr[Z_i = 1] = \Pr[X_i = 1] \cdot \Pr[Z_i = 1 \mid X_i = 1] = \frac{x_i}{4}(1 - \Pr[Z_i = 0 \mid X_i = 1]) \geq \frac{x_i}{16}.$$

□

One can improve the above analysis to show that $\Pr[Z_i = 1] \geq x_i/8$.

Theorem 10 *The randomized algorithm outputs a feasible solution of expected weight at least $\sum_{i=1}^{n} w_i x_i/16$.*

Proof: The expected weight of the output is

$$\mathbb{E}[\sum_i w_i x''_i] = \sum_i w_i \mathbb{E}[Z_i] \geq \sum_i w_i x_i/16$$

where we used the previous lemma to lower bound $\mathbb{E}[Z_i]$. □

Rounding for k-sparse PIPs: We now extend the rounding algorithm and analysis above to k-sparse PIPs. Let x be a feasible fractional solution to $\max\{wx \mid Ax \leq 1, x \in [0,1]^n\}$. For a column index i we let $N(i) = \{j \mid A_{j,i} > 0\}$ be the indices of the rows in which i has a non-zero entry. Since A is k-column-sparse we have that $|N(i)| \leq k$ for $1 \leq i \leq n$. When we have more than one constraint we cannot classify an item/index i as big or small since it may be big for some constraints and small for others. We say that i is small for constraint $j \in N(i)$ if $A_{j,i} \leq 1/2$ otherwise i is big for constraint j. Let $S_j = \{i \mid j \in N(i), \text{ and } i \text{ small for } j\}$ be the set of all small columns for j and $B_j = \{i \mid j \in N(i), \text{ and } i \text{ small for } j\}$ be the set of all big columns for j. Note that $S_j \cap B_j$ is the set of all i with $A_{j,i} > 0$.

ROUNDING-WITH-ALTERATION FOR k-SPARSE PIPS:
Let x be an optimum fractional solution
Round each i to 1 independently with probability $x_i/(4k)$. Let x' be rounded solution.
$x'' = x'$
For $j = 1$ to m do
  If ($x'_i = 1$ for exactly one $i \in B_j$)
    For each $h \in S_j \cup B_j$ and $h \neq i$ set $x''_h = 0$
  Else If ($\sum_{i \in S_j} A_{j,i} x'_i > 1$ *or* two or more items from B_j are chosen in x')
    For each $h \in S_j \cup B_j$ set $x''_h = 0$
  End If
End For
Output feasible solution x''

The algorithm, after picking the random solution x', alters it as follows: it applies the previous algorithm's strategy to each constraint j separately. Thus an element i can be rejected at different constraints $j \in N(i)$. We need to bound the total probability of rejection. As before, the following claim is easy to verify.

Claim 11 *The integer solution x'' is feasible.*

Now let us analyze the probability of an item i being present in the final solution. Let $\mathcal{E}_1(j)$ be the event that $\sum_{i \in S_j} A_{j,i} x'_i > 1$, that is the sum of the sizes of the items that are small for j in x' exceed the capacity. Let $\mathcal{E}_2(j)$ be the event that at least one big item for j is chosen in x'. The following claims follow from the same reasoning as the ones before with the only change being the scaling factor.

Claim 12 $\Pr[\mathcal{E}_1(j)] \leq 1/(4k)$.

Claim 13 $\Pr[\mathcal{E}_2(j)] \leq 1/(2k)$.

Lemma 14 *Let Z_i be the indicator random variable that is 1 if $x''_i = 1$ and 0 otherwise. Then* $\mathbb{E}[Z_i] = \Pr[Z_i = 1] \geq x_i/(16k)$.

Proof: We consider the binary random variable X_i which is 1 if $x'_i = 1$ after the randomized rounding. We have $\mathbb{E}[X_i] = \Pr[X_i = 1] = x_i/(4k)$. We write

$$\Pr[Z_i = 1] = \Pr[X_i = 1] \cdot \Pr[Z_i = 1 \mid X_i = 1] = \frac{x_i}{4k} \Pr[Z_i = 1 \mid X_i = 1].$$

We upper bound the probability $\Pr[Z_i = 0 | X_i = 1]$, that is, the probability that we reject i conditioned on the fact that it is chosen in the random solution x'. We observe that

$$\Pr[Z_i = 0 | X_i = 1] \leq \sum_{j \in N(i)} (\Pr[\mathcal{E}_1(j)] + \Pr[\mathcal{E}_2(j)] \leq k(1/(4k) + 1/(2k)) \leq 3/4.$$

We used the fact that $N(i) \leq k$ and the claims above. Therefore,

$$\Pr[Z_i = 1] = \Pr[X_i = 1] \cdot \Pr[Z_i = 1 \mid X_i = 1] = \frac{x_i}{4k}(1 - \Pr[Z_i = 0 \mid X_i = 1]) \geq \frac{x_i}{16k}.$$

□

The theorem below follows by using the above lemma and linearity of expectation to compare the expected weight of the output of the randomized algorithm with that of the fractional solution.

Theorem 15 *The randomized algorithm outputs a feasible solution of expected weight at least* $\sum_{i=1}^{n} w_i x_i/(16k)$. *There is* $1/(16k)$*-approximation for* k*-sparse PIPs.*

Larger width helps: We saw during the discussion on the KNAPSACK problem that if all items are small with respect to the capacity constraint then one can obtain better approximations. For PIPs we defined the *width* of a given instance as W if $\max_{i,j} A_{ij}/b_i \leq 1/W$; in other words no single item is more than $1/W$ times the capacity of any constraint. One can show using a very similar algorithm and anaylisis as above that the approximation bound improves to $\Omega(1/k^{\lceil W \rceil})$ for instance with with W. Thus if $W = 2$ we get a $\Omega(1/\sqrt{k})$ approximation instead of $\Omega(1/k)$-approximation. More generally when $W \geq c \log k/\epsilon$ for some sufficiently large constant c we can get a $(1-\epsilon)$-approximation.

References

[1] J. Håstad. Clique is Hard to Approximate within $n^{1-\epsilon}$. *Acta Mathematica*, 182:105–142, 1999.

[2] N. Bansal, N. Korula, V. Nagarajan, A. Srinivasan. On k-Column Sparse Packing Programs. *Proc. of IPCO*, 2010. Available at `http://arxiv.org/abs/0908.2256`.

CS 598CSC: Approximation Algorithms	Lecture date: February 18, 2011
Instructor: Chandra Chekuri	Scribe: CC

In the previous lecture we discussed *packing* problems of the form $\max\{wx \mid Ax \leq 1, x \in \{0,1\}^n\}$ where A is a non-negative matrix. In this lecture we consider "congestion minimization" in the presense of packing constraints. We address a routing problem that motivates these kinds of problems.

1 Chernoff-Hoeffding Bounds

For the analysis in the next section we need a theorem that gives quantitative estimates on the probability of deviating from the expectation for a random variable that is a sum of binary random variables.

Theorem 1 (Chernoff-Hoeffding) *Let $X_1, X_2, \ldots, X_n$ be independent binary random variables and let $a_1, a_2, \ldots, a_n$ be coefficients in $[0,1]$. Let $X = \sum_i a_i X_i$. Then*

- *For any $\mu \geq \mathbb{E}[X]$ and any $\delta > 0$, $\Pr[X > (1+\delta)\mu] \leq \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^\mu$.*
- *For any $\mu \leq \mathbb{E}[X]$ and any $\delta > 0$, $\Pr[X < (1-\delta)\mu] \leq e^{-\mu\delta^2/2}$.*

The bounds in the above theorem are what are called *dimension-free* in that the dependence is only on $\mathbb{E}[X]$ and not on n the number of variables.

The following corollary will be useful to us. In the statement below we note that m is not related to the number of variables n.

Corollary 2 *Under the conditions of the above theorem, there is a universal constant α such that for any $\mu \geq \max\{1, \mathbb{E}[X]\}$, and sufficiently large m and for $c \geq 1$, $\Pr[X > \frac{\alpha c \ln m}{\ln \ln m} \cdot \mu] \leq 1/m^c$.*

Proof: Choose δ such that $(1+\delta) = \frac{\alpha c \ln m}{\ln \ln m}$ for some sufficiently large constant α that we will specify later. Let m be sufficiently large such that $\ln \ln m - \ln \ln \ln m > (\ln \ln m)/2$. Now applying the upper tail bound in the first part of the above theorem for μ and δ, we have that

$$\begin{aligned}
\Pr[X > \frac{\alpha c \ln m}{\ln \ln m} \cdot \mu] &= \Pr[X > (1+\delta)\mu] \\
&\leq \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^\mu \\
&\leq \frac{e^\delta}{(1+\delta)^{(1+\delta)}} \quad \text{(since } \mu \geq 1 \text{ and the term inside is less than 1 for large } \alpha \text{ and } m\text{)} \\
&\leq \frac{e^{(1+\delta)}}{(1+\delta)^{(1+\delta)}} \\
&= (\frac{\alpha c \ln m}{e \ln \ln m})^{-\alpha c \ln m / \ln \ln m} \\
&= \exp((\ln \alpha c/e + \ln \ln m - \ln \ln \ln m)(-\alpha c \ln m / \ln \ln m)) \\
&\leq \exp(0.5 \ln \ln m (-\alpha c \ln m / \ln \ln m)) \quad (m \text{ and } \alpha \text{ are sufficiently large to ensure this}) \\
&\leq 1/m^{c\alpha/2} \leq 1/m^c \quad \text{(assuming } \alpha \text{ is larger than 2)}
\end{aligned}$$

□

2 Congestion Minimization for Routing

Let $G = (V, E)$ be a directed graph that represents a network on which traffic can be routed. Each edge $e \in E$ has a non-negative capacity $c(e)$. There are k pairs of nodes $(s_1, t_1), \ldots, (s_k, t_k)$ and each pair i is associated with a non-negative demand d_i that needs to be routed along a *single* path between s_i and t_i. In a first version we will assume that we are explicitly given for each pair i a set of paths $\mathcal{P}_i$ and the demand for i has to be routed along one of the paths in $\mathcal{P}_i$. Given a choice of the paths, say, $p_1, p_2, \ldots, p_k$ where $p_i \in \mathcal{P}_i$ we have an induced flow on each edge e. The flow on e is the total demand of all pairs whose paths contain e; that is $x(e) = \sum_{i:e \in p_i} d_i$. We define the congestion on e as $\max\{1, x(e)/c(e)\}$. In the congestion minimization problem the goal is to choose paths for the pairs to minimize the *maximum* congestion over all edges. We will make the following natural assumption. For any path $p \in \mathcal{P}_i$ and an edge $e \in p$, $c(e) \geq d_i$. One can write a linear programming relaxation for this problem as follows. We have variables $x_{i,p}$ for $1 \leq i \leq k$ and $p \in \mathcal{P}_i$ which indicate whether the path p is the one chosen for i.

$$\begin{aligned}
&\min \lambda \\
\text{subject to } &\sum_{p \in \mathcal{P}_i} x_{i,p} = 1 && 1 \leq i \leq k \\
&\sum_{i=1}^{k} d_i \sum_{p \in \mathcal{P}_i, e \in p} x_{i,p} \leq \lambda c(e) && e \in E \\
&x_{i,p} \geq 0 && 1 \leq i \leq k, p \in \mathcal{P}_i
\end{aligned}$$

Technically the objective function should be $\max\{1, \lambda\}$ which we can enforce by adding a constraint $\lambda \geq 1$.

Let λ^* be an optimum solution to the above linear program. It gives a lower bound on the optimum congestion. How do we convert a fractional solution to an integer solution? A simple randomized rounding algorithm was suggested by Raghavan and Thompson in their influential work [1].

RandomizedRounding:
Let x be an optimum fractional solution
For $i = 1$ to k do
  Independent of other pairs, pick a single path $p \in \mathcal{P}_i$ randomly such that $\Pr[p \text{ is chosen}] = x_{i,p}$

Note that for a given pair i we pick exactly one path. One can implement this step as follows. Since $\sum_{p \in \mathcal{P}_i} x_{i,p} = 1$ we can order the paths in $\mathcal{P}_i$ in some arbitrary fashion and partition the interval $[0, 1]$ by intervals of length $x_{i,p}$, $p \in \mathcal{P}_i$. We pick a number θ uniformly at random in $[0, 1]$ and the interval in which θ lies determines the path that is chosen.

Now we analyze the performance of the randomized algorithm. Let $X_{e,i}$ be a binary random variable that is 1 if the path chosen for i contains the edge e. Let $X_e = \sum_i d_i X_{e,i}$ be the total demand routed through e. We leave the proof of the following claim as an exercise to the reader.

Claim 3 $\mathbb{E}[X_{e,i}] = \Pr[X_{e,i} = 1] = \sum_{p \in \mathcal{P}_i, e \in p} x_{i,p}$.

The main lemma is the following.

Lemma 4 *There is a universal constant β such that $\Pr[X_e > \frac{\beta \ln m}{\ln \ln m} \cdot c(e) \max\{1, \lambda^*\}] \leq 1/m^2$ where m is the number of edges in the graph.*

Proof: Recall that $X_e = \sum_i d_i X_{e,i}$.

$$\mathbb{E}[X_e] = \sum_i d_i \mathbb{E}[X_{e,i}] = \sum_i d_i \sum_{p \in \mathcal{P}_i, e \in p} x_{i,p} \leq \lambda^* c(e).$$

The second equality follows from the claim above, and inequality follows from the constraint in the LP relaxation.

Let $Y_e = X_e / c(e) = \sum_i \frac{d_i}{c(e)} X_{e,i}$. From above $\mathbb{E}[Y_e] \leq \lambda^*$. The variables $X_{e,i}$ are independent since the paths for the different pairs are chosen independently. Y_e is a sum of independent binary random variables and each coefficient $d_i/c(e) \leq 1$ (recall the assumption). Therefore we can apply Chernoff-Hoeffding bounds and in particular Corollary 2 to Y_e with $c = 2$.

$$\Pr[Y_e \geq \frac{2\alpha \ln m}{\ln \ln m} \max\{1, \lambda^*\}] \leq 1/m^2.$$

The constant α above is the one guaranteed in Corollary 2. We can set $\beta = 2\alpha$. This proves the lemma by noting that $X_e = c(e) Y_e$. □

Theorem 5 *RandomizedRounding, with probability at least $(1 - 1/m)$ (here m is the number of edges) outputs a feasible integral solution with congestion upper bounded by $O(\frac{\ln m}{\ln \ln m}) \max\{1, \lambda^*\}$.*

Proof: From Lemma 4 for any fixed edge e, the probability of the congestion on e exceeding $\frac{\beta \ln m}{\ln \ln m} \max\{1, \lambda^*\}$ is at most $1/m^2$. Thus the probability that it exceeds this bound for *any* edge is at most $m \cdot 1/m^2 \leq 1/m$ by the union bounds over the m edges. Thus with probability at least $(1 - 1/m)$ the congestion on all edges is upper bounded by $\frac{\beta \ln m}{\ln \ln m} \max\{1, \lambda^*\}$. □

The above algorithm can be *derandomized* but it requires the technique of *pessimistic estimators* which was another innovation by Raghavan [2].

2.1 Unsplittable Flow Problem: when the paths are not given explicitly

We now consider the variant of the problem in which $\mathcal{P}_i$ is the set of *all* paths between s_i and t_i. The paths for each pair are not explicitly given to us as part of the input but only implicitly given. This problem is called the *unsplittable flow problem*. The main technical issue in extending the previous approach is that $\mathcal{P}_i$ can be exponential in n, the number of nodes. We cannot even write down the linear proogram we developed previously in polynomial time! However it turns out that one can in fact solve the linear program implicitly and find an optimal solution which has the added bonus of having polynomial-sized support; in other words the number of variables $x_{i,p}$ that are strictly positive will be polynomial. This should not come as a surprise since the linear program has only a polynomial number of non-trivial constraints and hence it has an optimum basic solution with small support. Once we have a solution with a polynomial-sized support the randomized rounding algorithm can be implemented in polynomial time by simply working with

those paths that have non-zero flow on them. How do we solve the linear program? This requires using the Ellipsoid method on the dual and then solving the primal via complementary slackness. We will discuss this at a later point.

A different approach is to solve a flow-based linear program which has the same optimum value as the path-based one. However, in order to implement the randomized rounding, one then has to decompose the flow along paths which is fairly standard in network flows. We now describe the flow based relaxation. We have variables $f(e,i)$ for each edge e and pair (s_i,t_i) which is the total amount of flow for pair i along edge e. We will send a unit of flow from s_i to t_i which corresponds to finding a path to route. In calculating congestion we will again scale by the total demand.

$$\begin{aligned}
&\min \lambda \\
\text{subject to } &\sum_{i=1}^{k} d_i f(e,i) \leq \lambda c(e) && e \in E \\
&\sum_{e \in \delta^+(s_i)} f(e,i) - \sum_{e \in \delta^-(s_i)} f(e,i) = 1 && 1 \leq i \leq k \\
&\sum_{e \in \delta^+(v)} f(e,i) - \sum_{e \in \delta^-(v)} f(e,i) = 0 && 1 \leq i \leq k, v \notin \{s_i,t_i\} \\
&f(e,i) \geq 0 && 1 \leq i \leq k, e \in E
\end{aligned}$$

The above linear program can be solved in polynomial time since it has only mk variables and $O(m+kn)$ constraints where m is the number of edges in the graph and k is the number of pairs. Given a feasible solution f for the above linear program we can, for each i, decompose the flow vector $f(.,i)$ for the pair (s_i,t_i) into flow along at most m paths. We then use these paths in the randomized rounding. For that we need the following flow-decomposition theorem for s-t flows.

Lemma 6 *Given a directed graph $G=(V,E)$ and nodes $s,t \in V$ and an s-t flow $f : E \to R_+$ there is a decomposition of f along s-t paths and cycles in G. More formally let $\mathcal{P}_{st}$ be the set of all s-t paths and let $\mathcal{C}$ be the set of directed cycles in G. Then there is a function $g : \mathcal{P}_{st} \cup \mathcal{C} \to R_+$ such that:*

- *For each e, $\sum_{q \in \mathcal{P}_{st} \cup \mathcal{C}, e \in q} g(q) = f(e)$.*
- *$\sum_{p \in \mathcal{P}_{st}} g(p)$ is equal to the value of the flow f.*
- *The support of g is at most m where m is the number of edges in G, that is, $|\{q | g(q) > 0\}| \leq m$. In particular, if f is acyclic $g(q) = 0$ for all $q \in \mathcal{C}$.*

Moreover, given f, g satisfying the above properties can be computed in polynomial time where the output consists only of paths and cycles with non-zero g value.

By applying the above ingredients we obtain the following.

Theorem 7 *There is an $O(\log n / \log \log n)$ randomized approximation algorithm for congestion minimization in the unsplittable flow problem.*

References

[1] P. Raghavan and C. D. Thompson. Randomized rounding: a technique for provably good algorithms and algorithmic proofs. *Combinatorica* 7(4):365–374, 1987.

[2] P. Raghavan. Probabilistic Construction of Deterministic Algorithms: Approximating Packing Integer Programs. *JCSS*, 37(2):130–143, 1988.

CS 598CSC: Approximation Algorithms	Lecture date: February 23 and 25, 2011
Instructor: Alina Ene	Scribe: Alina Ene

1 Scheduling on Unrelated Parallel Machines

We have a set J of n jobs, and a set M of m machines. The processing time of job i is p_{ij} on machine j. Let $f : J \leftarrow M$ be a function that assigns each job to exactly one machine. The *makespan* of f is $\max_{1 \leq j \leq m} \sum_{i:f(i)=j} p_{ij}$, where $\sum_{i:f(i)=j} p_{ij}$ is the total processing time of the jobs that are assigned to machine j. In the Scheduling on Unrelated Parallel Machines problem, the goal is to find an assignment of jobs to machines of minimum makespan.

We can write an LP for the problem that is very similar to the routing LP from the previous lecture. For each job i and each machine j, we have a variable x_{ij} that denotes whether job i is assigned to machine j. We also have a variable λ for the makespan. We have a constraint for each job that ensures that the job is assigned to some machine, and we have a constraint for each machine that ensures that the total processing time of jobs assigned to the machines is at most the makespan λ.

$$\begin{aligned}
\text{minimize} \quad & \lambda & \\
\text{subject to} \quad & \sum_{j \in M} x_{ij} = 1 & \forall i \in J \\
& \sum_{i \in J} x_{ij} p_{ij} \leq \lambda & \forall j \in M \\
& x_{ij} \geq 0 & \forall i \in J, j \in M
\end{aligned}$$

The above LP is very natural, but unfortunately it has unbounded integrality gap. Suppose that we have a single job that has processing time T on each of the machines. Clearly, the optimal schedule has makespan T. However, the LP can schedule the job to the extend of $1/m$ on each of the machines, i.e., it can set $x_{1j} = 1/m$ for all j, and the makespan of the resulting fractional schedule is only T/m.

To overcome this difficulty, we modify the LP slightly. Suppose we knew that the makespan of the optimal solution is equal to λ, where λ is some fixed number. If the processing time p_{ij} of job i on machine j is greater than λ, job i is *not* scheduled on machine j, and we can strengthen the LP by setting x_{ij} to 0 or equivalently, by removing the variable. More precisely, let $\mathcal{S}_\lambda = \{(i, j) \mid i \in J, j \in M, p_{ij} \leq \lambda\}$. Given a value λ, we can write the following LP for the problem.

$$\begin{aligned}
& \underline{\mathbf{LP}(\lambda)} & \\
& \sum_{j:\,(i,j) \in \mathcal{S}_\lambda} x_{ij} = 1 & \forall i \in J \\
& \sum_{i:\,(i,j) \in \mathcal{S}_\lambda} x_{ij} p_{ij} \leq \lambda & \forall j \in M \\
& x_{ij} \geq 0 & \forall (i, j) \in \mathcal{S}_\lambda
\end{aligned}$$

Note that the LP above does *not* have an objective function. In the following, we are only interested in whether the LP is feasible, i.e, whether there is an assignment that satisfies all the constraints. Also, we can think of λ as a parameter and $\mathbf{LP}(\lambda)$ as a family of LPs, one for each value of the parameter. A useful observation is that, if λ is a lower bound on the makespan of the optimal schedule, $\mathbf{LP}(\lambda)$ is feasible and it is a valid relaxation for the SCHEDULING ON UNRELATED PARALLEL MACHINES problem.

Lemma 1. *Let λ^* be the minimum value of the parameter λ such that $\mathbf{LP}(\lambda)$ is feasible. We can find λ^* in polynomial time.*

Proof: For any fixed value of λ, we can check whether $\mathbf{LP}(\lambda)$ is feasible using a polynomial-time algorithm for solving LPs. Thus we can find λ^* using binary search starting with the interval $[0, \sum_{i,j} p_{ij}]$. □

In the following, we will show how to round a solution to $\mathbf{LP}(\lambda^*)$ in order to get a schedule with makespan at most $2\lambda^*$. As we will see shortly, it will help to round a solution to $\mathbf{LP}(\lambda^*)$ that is a *vertex* solution.

Let x be a vertex solution to $\mathbf{LP}(\lambda^*)$. Let G be a bipartite graph on the vertex set $J \cup M$ that has an edge ij for each variable $x_{ij} \neq 0$. We say that job i is fractionally set if $x_{ij} \in (0, 1)$ for some j. Let F be the set of all jobs that are fractionally set, and let H be a bipartite graph on the vertex set $F \cup M$ that has an edge ij for each variable $x_{ij} \in (0, 1)$; note that H is the induced subgraph of G on $F \cup M$. As shown in Lemma 2, the graph H has a matching that matches every job in F to a machine, and we will use such a matching in the rounding algorithm.

Lemma 2. *The graph G has a matching that matches every job in F to a machine.*

We are now ready to give the rounding algorithm.

SUPM-Rounding
- Find λ^*
- Find a vertex solution x to $\mathbf{LP}(\lambda^*)$
- For each i and j such that $x_{ij} = 1$, assign job i to machine j
- Construct the graph H
- Find a maximum matching $\mathcal{M}$ in H
- Assign the fractionally set jobs according to the matching $\mathcal{M}$

Theorem 3. *Consider the assignment constructed by* SUPM-*Rounding. Each job is assigned to a machine, and the makespan of the schedule is at most $2\lambda^*$.*

Proof: By Lemma 2, the matching $\mathcal{M}$ matches every fractionally set job to a machine and therefore all of the jobs are assigned. After assigning all of the integrally set jobs, the makespan (of the partial schedule) is at most λ^*. Since $\mathcal{M}$ is a matching, each machine receives at most one additional job. Let i be a fractionally set job, and suppose that i is matched (in $\mathcal{M}$) to machine j. Since the pair (i, j) is in $\mathcal{S}_{\lambda^*}$, the processing time p_{ij} is at most λ^*, and therefore the total processing time of machine j increases by at most λ after assigning the fractionally set jobs. Therefore the makespan of the final schedule is at most $2\lambda^*$. □

Exercise: Give an example that shows that Theorem 3 is tight. That is, give an instance and a vertex solution such that the makespan of the schedule SUPM-Rounding is at least $(2 - o(1))\lambda^*$.

Since λ^* is a lower bound on the makespan of the optimal schedule, we get the following corollary.

Corollary 4. SUPM-*Rounding achieves a 2-approximation.*

Now we turn our attention to Lemma 2 and some other properties of vertex solutions to $\mathbf{LP}(\lambda)$.

Lemma 5. *If $\mathbf{LP}(\lambda)$ is feasible, any vertex solution has at most $m + n$ non-zero variables and it sets at least $n - m$ of the jobs integrally.*

Proof: Let x be a vertex solution to $\mathbf{LP}(\lambda)$. Let r denote the number of pairs in $\mathcal{S}_\lambda$. Note that $\mathbf{LP}(\lambda)$ has r variables, one for each pair $(i, j) \in \mathcal{S}_\lambda$. If x is a vertex solution, it satisfies r of the constraints of $\mathbf{LP}(\lambda)$ with equality. The first set of constraints consists of m constraints, and the second set of constraints consists of n constraints. Therefore at least $r - (m + n)$ of the tight constraints are from the third set of constraints, i.e., at least $r - (m + n)$ of the variables are set to zero.

We say that job i is set fractionally if $x_{ij} \in (0, 1)$ for some j; job i is set integrally if $x_{ij} \in \{0, 1\}$ for all j. Let I and F be the set of jobs that are set integrally and fractionally (respectively). Clearly, $|I| + |F| = n$. Any job i that is fractionally set is assigned (fractionally) to at least two machines, i.e., there exist $j \neq \ell$ such that $x_{ij} \in (0, 1)$ and $x_{i\ell} \in (0, 1)$. Therefore there are at least $2|F|$ distinct non-zero variables corresponding to jobs that are fractionally set. Additionally, for each job i that is integrally set, there is a variable x_{ij} that is non-zero. Thus the number of non-zero variables is at least $|I| + 2|F|$. Hence $|I| + |F| = n$ and $|I| + 2|F| \leq m + n$, which give us that $|I|$ is at least $n - m$. □

Definition 1. *A connected graph is a **pseudo-tree** if the number of edges it most the number of vertices plus one. A graph is a **pseudo-forest** if each of its connected components is a pseudo-tree.*

Lemma 6. *The graph G is a pseudo-forest.*

Proof: Let C be a connected component of G. We restrict $\mathbf{LP}(\lambda)$ and x to the jobs and machines in C to get $\mathbf{LP}'(\lambda)$ and x'. Note that x' is a feasible solution to $\mathbf{LP}'(\lambda)$. Additionally, x' is a vertex solution to $\mathbf{LP}'(\lambda)$. If not, x' is a convex combination of two feasible solutions x'_1 and x'_2 to $\mathbf{LP}'(\lambda)$. We can extend x'_1 and x'_2 to two solutions x_1 and x_2 to $\mathbf{LP}(\lambda)$ using the entries of x that are not in x'. By construction, x_1 and x_2 are feasible solutions to $\mathbf{LP}(\lambda)$. Additionally, x is a convex combination of x_1 and x_2, which contradicts the fact that x is a vertex solution. Thus x' is a vertex solution to $\mathbf{LP}'(\lambda)$ and, by Lemma 5, x' has at most $n' + m'$ non-zero variables, where n' and m' are the number of jobs and machines in C. Thus C has $n' + m'$ vertices and at most $n' + m'$ edges, and therefore it is a pseudo-tree. □

Proof of Lemma 2: Note that each job that is integrally set has degree one in G. We remove each integrally set job from G; note that the resulting graph is H. Since we removed an equal number of vertices and edges from G, it follows that H is a pseudo-forest as well. Now we construct a matching $\mathcal{M}$ as follows.

Note that every job vertex has degree at least 2, since the job is fractionally assigned to at least two machines. Thus all of the leaves (degree-one vertices) of H are machines. While H has at least one leaf, we add the edge incident to the leaf to the matching and we remove both of its endpoints from the graph. If H does not have any leaves, H is a collection of vertex-disjoint cycles, since it is a pseudo-forest. Moreover, each cycle has even length, since H is bipartite. We construct a perfect matching for each cycle (by taking alternate edges), and we add it to our matching. □

Exercise: (Exercise 17.1 in [3]) Give a proof of Lemma 2 using Hall's theorem.

2 Generalized Assignment Problem

The Generalized Assignment problem is a generalization of the Scheduling on Unrelated Parallel Machines problem in which there are costs associated with each job-machine pair, in addition to a processing time. More precisely, we have a set J of n jobs, a set M of m machines, and a target λ. The processing time of job i is p_{ij} on machine j, and the cost of assigning job i to machine j is c_{ij}. Let $f : J \to M$ be a function that assigns each job to exactly one machine. The assignment f is *feasible* if its makespan is at most λ (recall that λ is part of the input), and its cost is $\sum_i c_{if(i)}$. In the Generalized Assignment problem, the goal is to construct a minimum cost assignment f that is *feasible*, provided that there is a feasible assignment.

In the following, we will show that, if there is a schedule of cost C and makespan at most λ, then we can construct a schedule of cost at most C and makespan at most 2λ.

As before, we let $\mathcal{S}_\lambda$ denote the set of all pairs (i, j) such that $p_{ij} \leq \lambda$. We can generalize the relaxation $\mathbf{LP}(\lambda)$ from the previous section to the following LP.

GAP-LP

$$
\begin{aligned}
\text{min} \quad & \sum_{(i,j)\in\mathcal{S}_\lambda} x_{ij} c_{ij} \\
\text{subject to} \quad & \sum_{j:\,(i,j)\in\mathcal{S}_\lambda} x_{ij} = 1 && \forall i \in J \\
& \sum_{i:\,(i,j)\in\mathcal{S}_\lambda} x_{ij} p_{ij} \leq \lambda && \forall j \in M \\
& x_{ij} \geq 0 && \forall (i,j) \in \mathcal{S}_\lambda
\end{aligned}
$$

Since we also need to preserve the costs, we can no longer use the previous rounding; in fact, it is easy to see that the previous rounding is arbitrarily bad for the Generalized Assignment problem. However, we will still look for a matching, but in a slightly different graph.

But before we give the rounding algorithm for the Generalized Assignment problem, we take a small detour into the problem of finding a minimum-cost matching in a bipartite graph. In the Minimum Cost Biparite Matching problem, we are given a bipartite graph $B = (V_1 \cup V_2, E)$ with costs c_e on the edges, and we want to construct a minimum cost matching $\mathcal{M}$ that matches every vertex in V_1, if there is such a matching. For each vertex v, let $\delta(v)$ be the set of all edges incident to v. We can write the following LP for the problem.

BipartiteMatching(B)

$$\begin{aligned} \min \quad & \sum_{e \in E(B)} c_e y_e & \\ \text{subject to} \quad & \sum_{e \in \delta(v)} y_e = 1 & \forall v \in V_1 \\ & \sum_{e \in \delta(v)} y_e \leq 1 & \forall v \in V_2 \\ & y_e \geq 0 & \forall e \in E(B) \end{aligned}$$

Theorem 7 (Edmonds [2]). *For any bipartite graph B, any vertex solution to **BipartiteMatching**(B) is an integer solution. Moreover, given a feasible fractional solution, we can find in polynomial time a feasible solution z such that z is integral and*

$$\sum_{e \in E(B)} c_e z_e \leq \sum_{e \in E(B)} c_e y_e.$$

Let x be an optimal vertex solution to **GAP-LP**. As before, we want to construct a graph G that has a matching $\mathcal{M}$ that matches all jobs. The graph G will now have costs on its edges and we want a matching of cost at most C. Recall that for Scheduling on Unrelated Parallel Machines we defined a bipartite graph on the vertex set $J \cup M$ that has an edge ij for every variable x_{ij} that is non-zero. We can construct the same graph for Generalized Assignment, and we can assign a cost c_{ij} to each edge ij. If the solution x was actually a fractional matching — that is, if x was a feasible solution to **BipartiteMatching**(G) — Theorem 7 would give us the desired matching. The solution x satisfies the constraints corresponding to vertices $v \in J$, but it does not necessarily satisfy the constraints corresponding vertices $v \in M$, since a machine can be assigned more than one job. To get around this difficulty, we will introduce several nodes representing the same machine, and we will use x to construct a fractional matching for the resulting graph.

The fractional solution x assigns $\sum_{i \in J} x_{ij}$ jobs to machine j; let $k_j = \lceil \sum_{i \in J} x_{ij} \rceil$. We construct a bipartite graph G as follows. For each job i, we have a node i. For each machine j, we have k_j nodes $(j,1), \cdots, (j,k_j)$. We can think of the nodes $(j,1), \cdots, (j,k_j)$ as *slots* on machine j. Since now we have multiple slots on each of the machines, we need a fractional assignment y that assigns a job to slots on the machines. More precisely, y has an entry $y_{i,(j,s)}$ for each job i and each slot (j,s) that represents the fraction of job i that is assigned to the slot. We give the algorithm that constructs y from x below. Once we have the solution y, we add an edge between any job i and any machine slot (j,s) such that $y_{i,(j,s)}$ is non-zero. Additionally, we assign a cost $c_{i,(j,s)}$ to each edge $(i,(j,s))$ of G that is equal to c_{ij}.

When we construct y, we consider each machine in turn. Let j be the current machine. Recall that we want to ensure that y assigns at most one job to each slot; as such, we will think of each slot on machine j as a bin with capacity 1. We "pack" jobs into the bins greedily. We only consider jobs i such that p_{ij} is at most λ; let h denote the number of such jobs. We assume without loss of generality that these are labeled as $1, 2, \cdots, h$, and $p_{1j} \geq p_{2j} \geq \cdots \geq p_{hj}$. Informally, when we construct y, we consider the jobs $1, 2, \cdots, h$ in this order. Additionally, we keep track of the bin that has not been filled and the amount of space s available on that bin. When we consider job i,

```
GreedyPacking(x)
  y = 0 ⟨⟨initialize y to 0⟩⟩
  s = 1 ⟨⟨s is the current bin⟩⟩
  R = 1 ⟨⟨R is the space available on bin s⟩⟩
  for i = 1 to h
  ⟨⟨pack x_ij into the bins⟩⟩
  if x_ij ≤ R
       y_{i,(j,s)} = x_ij
       R = R − x_ij
       if R = 0
            s = s + 1
            R = 1
  else
       y_{i,(j,s)} = R
       y_{i,(j,s+1)} = x_ij − R ⟨⟨pack x_ij − R in the next bin⟩⟩
       R = 1 − y_{i,(j,s+1)}
       s = s + 1
  return y
```

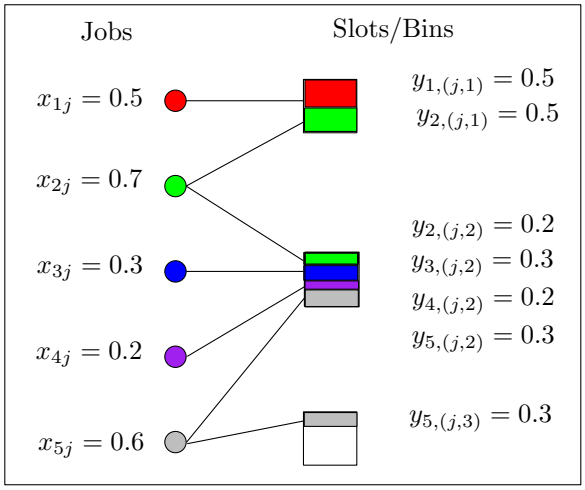


Figure 1. Constructing y from x.

we try to pack x_{ij} into the current bin: if there is at least x_{ij} space available, i.e., $x_{ij} \leq s$, we pack the entire amount into the current bin; otherwise, we pack as much as we can into the current bin, and we pack the rest into the next bin. (See Figure 1 for an example.)

Lemma 8. *The solution y constructed by* GreedyPacking *is a feasible solution to* BipartiteMatching(G). *Moreover,*

$$\sum_{(i,(j,s))\in E(G)} y_{i,(j,s)} c_{i,(j,s)} = \sum_{(i,j)\in\mathcal{S}_\lambda} x_{ij} c_{ij}.$$

Proof: Note that, by construction, $x_{ij} = \sum_{s=1}^{k_j} y_{i,(j,s)}$. Therefore, for any job i, we have

$$\sum_{(i,(j,s))\in\delta(i)} y_{i,(j,s)} = \sum_{j:\ (i,j)\in\mathcal{S}_\lambda} \sum_{s=1}^{k_j} y_{i,(j,s)} = \sum_{j:\ (i,j)\in\mathcal{S}_\lambda} x_{ij} = 1$$

Additionally, since we imposed a capacity of 1 on the bins associated with each slot, it follows that, for any slot (j,s),

$$\sum_{(i,(j,s)\in\delta((j,s)))} y_{i,(j,s)} \leq 1$$

Therefore y is a feasible solution to BipartiteMatching(G). Finally,

$$\sum_{(i,(j,s))\in E(G)} = \sum_{i=1}^{n} \sum_{j:\ (i,j)\in\mathcal{S}_\lambda} \sum_{s=1}^{k_j} y_{i,(j,s)} c_{ij} = \sum_{(i,j)\in\mathcal{S}_\lambda} x_{ij} c_{ij}$$

□

Theorem 7 gives us the following corollary.

Corollary 9. *The graph G has a matching $\mathcal{M}$ that matches every job and it has cost at most $\sum_{(i,j)\in\mathcal{S}_\lambda} x_{ij}c_{ij}$. Moreover, we can find such a matching in polynomial time.*

GAP-Rounding
let x be an optimal solution to **GAP-LP**
$y = \text{GREEDYPACKING}(x)$
construct the graph G
construct a matching $\mathcal{M}$ in G such that $\mathcal{M}$ matches every job
    and the cost of $\mathcal{M}$ is at most $\sum_{(i,j)\in\mathcal{S}_\lambda} x_{ij}c_{ij}$
for each edge $(i,(j,s)) \in \mathcal{M}$
    assign job i to machine j

Theorem 10. *Let $C = \sum_{(i,j)\in\mathcal{S}_\lambda} x_{ij}c_{ij}$. The schedule returned by **GAP-Rounding** has cost at most C and makespan at most 2λ.*

Proof: By Corollary 9, the cost of the schedule is at most C. Therefore we only need to upper bound the makespan of the schedule.

Consider a machine j. For any slot (j,s) on machine j, let

$$q_{js} = \max_{i:y_{i,(j,s)}>0} p_{ij}$$

That is, q_{js} is the maximum processing time of any pair ij such that job i is assigned (in y) to the slot (j,s). It follows that the total processing time of the jobs that $\mathcal{M}$ assigns to machine j is at most $\sum_{s=1}^{k_j} q_{js}$.

Since **GAP-LP** has a variable x_{ij} only for pairs (i,j) such that p_{ij} is at most λ, it follows that q_{j1} is at most λ. Therefore we only need to show that $\sum_{s=2}^{k_j} q_{js}$ is at most λ as well. Consider a slot s on machine j such that $s > 1$. Recall that we labeled the jobs that are relevant to machine j — that is, jobs i such that p_{ij} is at most λ — as $1, 2, \cdots, h$ such that $p_{1j} \geq p_{2j} \geq \cdots \geq p_{hj}$. Consider a job ℓ that is assigned to slot s. Since GREEDYPACKING considers jobs in non-increasing order according to their processing times, the processing time $p_{\ell j}$ of job ℓ is at most the processing time of any job assigned to the slot $s-1$. Therefore $p_{\ell j}$ is upper bounded by any convex combination of the processing times of the jobs that are assigned to the slot $s-1$. Since the slot $s-1$ is full, $\sum_i y_{i,(j,s-1)} = 1$ and thus $p_{\ell j}$ is at most $\sum_i y_{i,(j,s-1)}p_{ij}$. It follows that

$$\sum_{s=2}^{k_j} q_{js} \leq \sum_{s=2}^{k_j}\sum_i y_{i,(j,s-1)}p_{ij} \leq \sum_{s=1}^{k_j}\sum_i y_{i,(j,s)}p_{ij}$$

By construction, $\sum_s y_{i,(j,s)} = x_{ij}$, and therefore

$$\sum_{s=1}^{k_j}\sum_i y_{i,(j,s)}p_{ij} = \sum_i p_{ij}\sum_{s=1}^{s} y_{i,(j,s)} = \sum_i p_{ij}x_{ij}$$

Since x is a feasible solution to the **GAP-LP**,

$$\sum_{s=2}^{k_j} q_{js} \leq \sum_i p_{ij}x_{ij} \leq \lambda$$

which completes the proof. □

References

[1] Chandra Chekuri, *Approximation Algorithms Lecture Notes*, Lecture 12, http://www.cs.uiuc.edu/~chekuri/teaching/fall2006/lect12.pdf, 2006.

[2] Alexander Schrijver, *Combinatorial Optimization*, Chapter 17, Springer Verlag, 2003.

[3] Vijay Vazirani, *Approximation Algorithms*, Chapter 17, Springer Verlag, 2001.

[4] David Williamson and David Shmoys, *The Design of Approximation Algorithms*, Chapter 11, http://www.designofapproxalgs.com/, 2010.

CS 598CSC: Approximation Algorithms	Lecture date: March 2, 2011
Instructor: Chandra Chekuri	Scribe: CC

Local search is a powerful and widely used heuristic method (with various extensions). In this lecture we introduce this technique in the context of approximation algorithms. The basic outline of local search is as follows. For an instance I of a given problem let $\mathcal{S}(I)$ denote the set of feasible solutions for I. For a solution S we use the term *(local) neighborhood* of S to be the set of all solution S' such that S' can be obtained from S via some *local moves*. We let $N(S)$ denote the neighborhood of S.

```
LocalSearch:
Find a "good" initial solution S_0 ∈ S(I)
S ← S_0
repeat
  If (∃S' ∈ N(S) such that val(S') is strictly better than val(S))
      S ← S'
  Else
      S is a local optimum
      return S
  EndIf
Until (True)
```

For minimization problems S' is strictly better than S if $\text{val}(S') < \text{val}(S)$ whereas for maximization problems it is the case if $\text{val}(S') > \text{val}(S)$.

The running time of the generic local search algorithm depends on several factors. First, we need an algorithm that given a solution S either declares that S is a local optimum or finds a solution $S' \in N(S)$ such that $\text{val}(S')$ is strictly better thatn $\text{val}(S)$. A standard and easy approach for this is to ensure that the local moves are defined in such a way that $|N(S)|$ is polynomial in the input size $|I|$ and $N(S)$ can be enumerated efficiently; thus one can check each $S' \in N(S)$ to see if any of them is an improvement over S. However, in some more advanced settings, $N(S)$ may be exponential in the input size but one may be able to find a solution in $S' \in N(S)$ that improves on S in polynomial time. Second, the running time of the algorithm depends also on the number of iterations it takes to go from S_0 to a local optimum. In the worst case the number of iterations could be $|\text{OPT} - \text{val}(S_0)|$ which need not be strongly polynomial in the input size. We will see that one can often use a standard scaling trick to overcome this issue; basically we stop the algorithm unless the improvement obtained over the current S is a significant fraction of $\text{val}(S)$. Finally, the quality of the initial solution S_0 also factors into the running time.

1 Local Search for Max Cut

We illustrate local search for the well-known Max Cut problem. In Max Cut we are given an undirected graph $G = (V, E)$ and the goal is to partition V into $(S, V \setminus S)$ so as to maximize the number of edges crossing S, that is, $|\delta_G(S)|$. In the weighted version each edge e has a non-negative weight $w(e)$ the goal is to maximize the weight of the edges crossing S, that is, $w(\delta_G(S))$ where $w(A) = \sum_{e \in A} w(e)$.

We consider a simple local search algorithm for MAX CUT that starts with an arbitrary set $S \subseteq V$ and in each iteration either adds a vertex to S or removes a vertex from S as long as it improves the cut capacity.

```
LocalSearch for Max Cut:
S ← ∅
repeat
  If (∃v ∈ V \ S such that w(δ(S + v)) > w(δ(S)))
      S ← S + v
  Else If (∃v ∈ S such that w(δ(S − v)) > w(δ(S)))
      S ← S − v
  Else
      S is a local optimum
      return S
  EndIf
Until (True)
```

We will first focus on the quality of solution output by the local search algorithm.

Lemma 1 *Let S be a local optimum outbut the local search algorithm. Then for each vertex v, $w(\delta(S) \cap \delta(v)) \geq w(\delta(v))/2$.*

Proof: Let $\alpha_v = w(\delta(S) \cap \delta(v))$ be the weight of edges among those incident to v ($\delta(v)$) that cross the cut S. Let $\beta_v = w(\delta(v)) - \alpha_v$.

We claim that $\alpha_v \geq beta_v$ for each v. If $v \in V \setminus S$ and $\alpha_v < \beta_v$ then moving v to S will strictly increase $w(\delta(S))$ and S cannot be a local optimum. Similarly if $v \in S$ and $\alpha_v < \beta_v$ $w(\delta(S - v)) > w(\delta(S))$ and S is not a local optimum. □

Corollary 2 *If S is a local optimum then $w(\delta(S)) \geq w(E)/2 \geq \mathrm{OPT}/2$.*

Proof: Since each edge is incident to exactly two vertices we have $w(\delta(S)) = \frac{1}{2}\sum_{v \in V} w(\delta(S) \cap \delta(v))$. Apply the above lemma,

$$\begin{aligned} w(\delta(S)) &= \frac{1}{2}\sum_{v \in V} w(\delta(S) \cap \delta(v)) \\ &\geq \frac{1}{2}\sum_{v \in V} w(\delta(v))/2 \\ &\geq \frac{1}{2} w(E) \\ &\geq \frac{1}{2}\mathrm{OPT}, \end{aligned}$$

since $\mathrm{OPT} \leq w(E)$. □

The running time of the local search algorithm depends on the number of local improvement iterations; checking whether there is a local move that results in an improvement can be done by trying all possible vertices. If the graph is unweighted then the algorithm terminates in at most $|E|$ iterations. However, in the weighted case, it is known that the algorithm can take an exponential

time in $|V|$ when the weights are large. Many local search algorithms can be modified slightly to terminate with an approximate local optimum such that (i) the running time of the modified algorithm is strongly polynomial in the input size and (ii) the quality of the solution is very similar to that given by the original local search. We illustrate these ideas for MAX CUT. Consider the following algorithm where $\epsilon > 0$ is a parameter that can be chosen. Let n be the number of nodes in G.

```
MODIFIED LOCALSEARCH FOR MAX CUT(ε):
S ← {v*} where v* = arg max_{v∈V} w(δ(v))
repeat
  If (∃v ∈ V \ S such that w(δ(S + v)) > (1 + ε/n)w(δ(S)))
      S ← S + v
  Else If (∃v ∈ S such that w(δ(S − v)) > (1 + ε/n)w(δ(S)))
      S ← S − v
  Else
      return S
  EndIf
Until (True)
```

The above algorithm terminates unless the improvement is a relative factor of $(1 + \frac{\epsilon}{n})$ over the current solution's value. Thus the final output S is an *approximate* local optimum.

Lemma 3 *Let S be the output of the modified local search algorithm for* MAX CUT. *Then $w(\delta(S)) \geq \frac{1}{2(1+\epsilon/4)} w(E)$.*

Proof: As before let $\alpha_v = w(\delta(S) \cap \delta(v))$ and $\beta_v = w(\delta(v)) - \alpha_v$. Since S is an approximately local optimum we claim that for each v

$$\beta_v - \alpha_v \leq \frac{\epsilon}{n} w(\delta(S)).$$

Otherwise a local move using v would improve S by more than $(1 + \epsilon/n)$ factor. (The formal proof is left as an exercise to the reader).

We have,

$$\begin{aligned}
w(\delta(S)) &= \frac{1}{2} \sum_{v \in V} \alpha_v \\
&= \frac{1}{2} \sum_{v \in V} ((\alpha_v + \beta_v) - (\beta_v - \alpha_v))/2 \\
&\geq \frac{1}{4} \sum_{v \in V} (w(\delta(v)) - \frac{\epsilon}{n} w(S)) \\
&\geq \frac{1}{2} w(E) - \frac{1}{4} \sum_{v \in V} \frac{\epsilon}{n} w(S) \\
&\geq \frac{1}{2} w(E) - \frac{1}{4} \epsilon \cdot w(S).
\end{aligned}$$

Therefore $w(S)(1 + \epsilon/4) \geq w(E)/2$ and the lemma follows. □

Now we argue about the number of iterations of the algorithm.

Lemma 4 *The modified local search algorithm terminates in $O(\frac{1}{\epsilon} n \log n)$ iterations of the improvement step.*

Proof: We observe that $w(S_0) = w(\delta(v^*)) \geq \frac{1}{2n} w(E)$ (why?). Each local improvement iteration improves $w(\delta(S))$ by a multiplicative factor of $(1 + \epsilon/n)$. Therefore if k is the number of iterations that the algorithm runs for then $(1 + \epsilon/n)^k w(S_0) \leq w(\delta(S_)$ where S is the final output. However, $w(\delta(S)) \leq w(E)$. Hence

$$(1 + \epsilon/n)^k w(E)/2n \leq w(E)$$

which implies that $k = O(\frac{1}{\epsilon} n \log n)$. □

A tight example for local optimum: Does the local search algorithm do better than 1/2? Here we show that a local optimum is no better than a 1/2-approximation. Consider a complete bipartite graph $K_{2n,2n}$ with $2n$ vertices in each part. If L and R are the parts a set S where $|S \cap L| = n = |S \cap R|$ is a local optimum with $|\delta(S)| = |E|/2$. The optimum solution for this instance is $|E|$.

Max Directed Cut: A problem related to Max Cut is Max Directed Cut in which we are given a directed edge-weighted graph $G = (V, E)$ and the goal is to find a set $S \subseteq V$ that maximizes $w(\delta^+_G(S))$; that is, the weight of the directed edges leaving S. One can apply a similar local search as the one for Max Cut. However, the following example shows that the output S can be arbitrarily bad. Let $G = (V, E)$ be a directed in-star with center v and arcs connecting each of $v_1, \ldots, v_n$ to v. Then $S = \{v\}$ is a local optimum with $\delta^+(S) = \emptyset$ while OPT $= n$. However, a minor tweak to the algorithm gives a 1/3-approximation! Instead of returning the local optimum S return the better of S and $V \setminus S$. This step is needed because the directed cuts are not symmetric.

2 Local Search for Submodular Function Maximization

In this section we consider the utility of local search for maximizing non-negative submodular functions. Let $f : 2^V \to \mathbb{R}_+$ be a *non-negative* submodular set function on a ground set V. Recall that f is submodular if $f(A) + f(B) \geq f(A \cup B) + f(A \cap B)$ for all $A, B \subseteq V$. Equivalently f is submodular if $f(A + v) - f(A) \geq f(B + v) - f(B)$ for all $A \subset B$ and $v \notin B$. f is *monotone* if $f(A) \leq f(B)$ for all $A \subseteq B$. f is *symmetric* if $f(A) = f(V \setminus A)$ for all $A \subseteq V$. Submodular functions arise in a number of settings in combinatorial optimization. Two important examples are the following.

Example: Coverage in set systems. Let $S_1, S_2, \ldots, S_n$ be subsets of a set $\mathcal{U}$. Let $V = \{1, 2, \ldots, n\}$ and define $f : 2^V \to \mathbb{R}_+$ where $f(A) = |\cup_{i \in A} S_i|$. f is a monotone submodular function. One can also associate weights to elements of $\mathcal{U}$ via a function $w : \mathcal{U} \to \mathbb{R}_+$; the function f defined as $f(A) = w(\cup_{i \in A} S_i)$ is also monotone submodular.

Example: Cut functions in graphs. Let $G = (V, E)$ be an undirected graph with non-negative edge weights $w : E \to \mathbb{R}_+$. The cut function $f : 2^V \to \mathbb{R}_+$ defined as $f(S) = \sum_{e \in \delta_G(S)} w(e)$ is a symmetric submodular function; it is not monotone unless the graph is trivial. If G is directed and we define f as $f(S) = \sum_{e \in \delta^+_G(S)} w(e)$ then f is submodular but is not necessarily symmetric.

The following problem generalizes MAX CUT and MAX DIRECTED CUT that we have already seen.

Problem: MAX SUBMOD FUNC. Given a non-negative submodular set function f on a ground set V via a *value oracle*[1] find $\max_{S \subseteq V} f(S)$.

Note that if f is monotone then the problem is trivial since V is the optimum solution. Therefore, the problem is interesting (and NP-Hard) only when f is not necessarily monotone. We consider a simple local search algorithm for MAX SUBMOD FUNC and show that it gives a 1/3-approximation and a 1/2-approximation when f is symmetric. This was shown in [2].

LOCALSEARCH FOR MAX SUBMOD FUNC:
$S \leftarrow \emptyset$
repeat
  If ($\exists v \in V \setminus S$ such that $f(S+v) > f(S)$)
    $S \leftarrow S + v$
  Else If ($\exists v \in S$ such that $f(S-v) > f(S)$)
    $S \leftarrow S - v$
  Else
    S is a *local optimum*
    return the better of S and $V \setminus S$
  EndIf
Until (True)

We start the analysis of the algorithm with a basic lemma on submodularity.

Lemma 5 *Let $f : 2^V \rightarrow \mathbb{R}_+$ be a submodular set function on V. Let $A \subset B \subseteq V$. Then*

- *If $f(B) > f(A)$ then there is an element $v \in B \setminus A$ such that $f(A+v) - f(A) > 0$. More generally there is an element $v \in B \setminus A$ such that $f(A+v) - f(A) \geq \frac{1}{|B \setminus A|}(f(B) - f(A))$.*
- *If $f(A) > f(B)$ then there is an element $v \in B \setminus A$ such that $f(B-v) - f(B) > 0$. More generally there is an element $v \in B \setminus A$ such that $f(B-v) - f(B) \geq \frac{1}{|B \setminus A|}(f(A) - f(B))$.*

We obtain the following corollary.

Corollary 6 *Let S be a local optimum for the local search algorithm and let S^* be an optimum solution. Then $f(S) \geq f(S \cap S^*)$ and $f(S) \geq f(S \cup S^*)$.*

Theorem 7 *The local search algorithm is a 1/3-approximation and is a 1/2-approximation if f is symmetric.*

Proof: Let S be the local optimum and S^* be a global optimum for the given instance. From the previous corollary we have that $f(S) \geq f(S \cap S^*)$ and $f(S) \geq f(S \cup S^*)$. Note that the algorithm outputs the better of S and $V \setminus S$. By submodularity, we have,

$$f(V \setminus S) + f(S \cup S^*) \geq f(S^* \setminus S) + f(V) \geq f(S^* \setminus S)$$

[1] A value oracle for a set function $f : 2^V \rightarrow \mathbb{R}$ provides access to the function by giving the value $f(A)$ when presented with the set A.

where we used the non-negativity of f in the second inequality. Putting together the inequalities,

$$\begin{aligned} 2f(S) + f(V \setminus S) &= f(S) + f(S) + f(V \setminus S) \\ &\geq f(S \cap S^*) + f(S^* \setminus S) \\ &\geq f(S^*) + f(\emptyset) \\ &\geq f(S^*) = \text{OPT}. \end{aligned}$$

Thus $2f(S) + f(V \setminus S) \geq \text{OPT}$ and hence $\max\{f(S), f(V \setminus S)\} \geq \text{OPT}/3$.

If f is symmetric we argue as follows. Using Lemma 5 we claim that $f(S) \geq f(S \cap S^*)$ as before but also that $f(S) \geq f(S \cup \bar{S}^*)$ where $\bar{A}$ is shorthand notation for the the complement $V \setminus A$. Since f is symmetric $f(S \cup \bar{S}^*) = f(V \setminus (S \cup \bar{S}^*)) = f(\bar{S} \cap S^*) = f(S^* \setminus S)$. Thus,

$$\begin{aligned} 2f(S) &\geq f(S \cap S^*) + f(S \cup \bar{S}^*) \\ &\geq f(S \cap S^*) + f(S^* \setminus S) \\ &\geq f(S^*) + f(\emptyset) \\ &\geq f(S^*) = \text{OPT}. \end{aligned}$$

Therefore $f(S) \geq \text{OPT}/2$. □

The running time of the local search algorithm may not be polynomial but one can modify the algorithm as we did for Max Cut to obtain a strongly polynomial time algorithm that gives a $(1/3 - o(1))$-approximation ($(1/2 - o(1)$ for symmetric). See [2] for more details. There has been much work on submodular function maximization including work on variants with additional constraints. Local search has been a powerful tool in these algorithms. See references for some of these results and further pointers.

References

[1] C. Chekuri, J. Vondrák, and R. Zenklusen. Submodular function maximization via the multilinear relaxation and contention resolution schemes. *Proc. of ACM STOC*, 2001. To appear.

[2] U. Feige, V. Mirrokni and J. Vondrák. Maximizing a non-monotone submodular function. *Proc. of IEEE FOCS*, 461–471, 2007.

[3] J. Lee, V. Mirrokni, V. Nagarajan and M. Sviridenko. Maximizing nonmonotone submodular functions under matroid and knapsack constraints. *SIAM J. on Disc. Math.*, 23(4): 2053–2078, 2010. Preliminary version in *Proc. of ACM STOC*, 323–332, 2009.

[4] S. Oveis Gharan and J. Vondrák. Submodular maximization by simulated annealing. *Proc. of ACM-SIAM SODA*, 1098–1116, 2011.