Limits of Approximation Algorithms 2 Feb, 2010 (IMSc)

Lec. 1: Approximation Algorithms for NP-hard problems

Lecturer: Prahladh Harsha *Scribe: Srikanth Srinivasan*

In this course, we will be studying, as the title suggests, the approximability and inapproximability (limits of approximability) of different combinatorial optimization problems. All the problems we will be looking at will be ones that lack efficient algorithms and in particular will be NP-hard problems. The last two-three decades has seen remarkable progress in approximation algorithms for several of these NP-hard problems. The theory of NP-completeness provides a satisfactory theory of algorithmic hardness in many ways, however it is unable to explain the vastly different approximabilities of different NP-hard problems. Since the early 90's, work on probabilistic proof systems have shed light on the limitations to approximation algorithms. We will study in detail this connection between proof systems and (in)approximability and spend considerable time on the construction of such proof systems. In the first few lectures, we will discuss several approximation algorithms before we proceed to the topic of the course, "limits of approximation algorithm". This course will be a fleshed out version of a recent DIMACS tutorial by the same name [HC09].

The agenda for this first lecture is as follows.

- A brief history of Combinatorial Optimization.
- A brief survey of the area of Approximation Algorithms, with some examples.

The references for this lecture include Lecture 1 of the DIMACS tutorial [HC09], Lectures 1 and 2 of Sudan's course on inapproximability at MIT [Sud99], and Sections 1.1 and 3.2 from Vazirani's book on Approximation Algorithms [Vaz04].

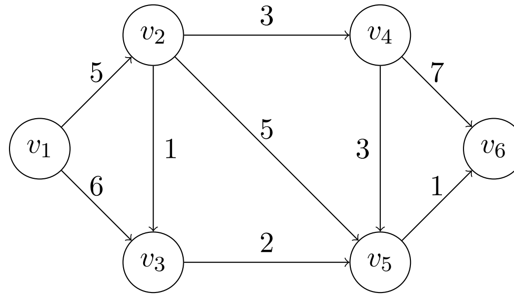
1.1 Combinatorial Optimization

The area of Combinatorial Optimization deals with algorithmic problems of the following flavour: Find a "best" object in a possibly large, but finite, space. As a subfield of mathematics, this area is relatively new, having been studied only in the last 100 years or so. One of the early references is the book of Lawler (1976) titled "Combinatorial Optimization: Matroids and Networks".

We begin with possibly two of the most prominent examples of Combinatorial Optimization problems: Network Flows, and Linear Programming.

1.1.1 Network Flows

The input in this problem consists of the following: a directed graph $G = (V, E)$, a designated source s and sink t of G, and a capacity function $c : E \to \mathbb{Z}^+$ defined on the edges of G.



In the *Flow Problem*, we are asked to compute the maximum amount of flow one can route from the source s to sink t. More formally, we need to find a *flow* $f : E \to \mathbb{R}^+$ such that:

- $\forall e \in E, \quad f(e) \le c(e)$,
- $\forall v \notin \{s, t\}, \quad \sum_{e \text{ into } v} f(e) = \sum_{e \text{ leaving } v} f(e)$, and
- The quantity $\left(\sum_{e \text{ leaving } s} f(e) - \sum_{e \text{ into } s} f(e)\right)$ is maximized, among all flows satisfying the above two conditions.

In the form stated stated above, it is not clear that the flow problem is a combinatorial optimization problem, since it involves searching for solutions in an infinite space (the space of all possible flows). However, by the *Max-flow Min-cut theorem* of Ford and Fulkerson, it is known that there is an optimum integer-valued flow in the network and thus the problem is really only a search problem over a finite space – a combinatorial optimization problem!. In fact, Ford and Fulkerson designed a combinatorial algorithm for this problem based on the Max-Flow Min-Cut theorem.

1.1.2 Linear Programming (LP)

In this case, we want to solve optimization problems of the following form:

$$\begin{aligned} & \max c^{\mathrm{T}} x \\ \text{subject to} \quad & Ax \le b \\ \text{where} \quad & A \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m, c \in \mathbb{R}^n \end{aligned}$$

Again, the above problem does not really look like a Combinatorial Optimization problem, given that our search space (a priori) is the Euclidean space $\mathbb{R}^n$. However, it is known (by

Dantzing and von Neumann) that if the above optimization problem has an optimum at all, there is a vertex of the polyhedron described by the system of inequalities that achieves this optimum. Hence, the problem boils down to searching for the optimum among the vertices of the polyhedron, of which there are only finitely many. Based on this idea, Dantzig designed the *Simplex algorithm* for Linear Programming.

1.1.3 A more formal view of Combinatorial Optimization problems

We will formally define a combinatorial optimization problem Π to be a 4-tuple $(\mathcal{X}, \mathcal{Y}, \text{Feas}, \text{Objective})$ where $\mathcal{X}$ refers to an input space, $\mathcal{Y}$ a solution space $\mathcal{Y}$, $\text{Feas} : \mathcal{X} \times \mathcal{Y} \to \{0, 1\}$ a feasibility function and $\text{Objective} : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$ the objective function.

Given an input instance $x \in \mathcal{X}$, we say that a solution $y \in \mathcal{Y}$ is *feasible* if $\text{Feas}(x, y) = 1$. The focus of the problem is related to finding those $y^* \in \mathcal{Y}$ that are feasible and maximize (or minimize) the Objective function $\text{Objective}(x, y)$ among all feasible y: such a y^* is called *optimum*; we denote by $\text{OPT}(x)$ the value $\text{Objective}(x, y^*)$.

Fix $\mathcal{X}, \mathcal{Y}, \text{Feas}$, and Objective as above. Associated with them are three related algorithmic problems:

1. The Search Problem: Given as input $x \in \mathcal{X}$, find an optimum y^*. This is the kind of problem described in the examples above.
2. The Computational Problem: Given $x \in \mathcal{X}$, find $\text{OPT}(x)$.
3. The Decision Problem: Given $x \in \mathcal{X}$ and $k \in \mathbb{R}$, is $\text{OPT}(x) \geq k$?

It is easily seen that the above problems are listed in nonincreasing order of hardness. That is, if the search problem is solvable efficiently, then so is the computational problem, and if the computational problem is efficiently solvable, then so is the decision problem.

1.2 Efficiency and the theory of NP-completeness

The two algorithms that we mentioned above – the original algorithm of Ford and Fulkerson, and Dantzig's Simplex algorithm – solve the problems that they are supposed to, but aren't *efficient* in today's sense of the word. In an important paper in 1965, Edmonds introduced the notion of P – the class of decision problems that can be solved in polynomial time – and argued that this was an interesting notion of "efficient" algorithm. Throughout this course, by an efficient algorithm, we will mean an algorithm (sometimes randomized) that runs in time polynomial in the size of its input. Neither Simplex nor the orignal Ford-Fulkerson algorithm are polynomial time, though the problems that they solve were proved to be in P via other algorithms.

However, there were many interesting algorithmic problems for which such efficient solutions remained elusive. To explain coherently why many natural problems did *not* seem to have a polynomial-time solution, Cook, Levin, and Karp introduced the theory of NP-completeness. The class NP is another class of decision problems and is defined as follows.

Definition 1.2.1. *A language $L \subseteq \{0,1\}^*$ is in* NP *iff there is a polynomial-time algorithm M and a polynomial $p(\cdot)$ such that for every $x \in \{0,1\}^*$, $x \in L$ iff there is a $y \in \{0,1\}^{p(|x|)}$ such that $M(x,y) = 1$.*

Cook defined the class NP in 1971 and showed that the decision problem of checking satisfiability of a given boolean formula is "as hard as" any other problem in this class. To state this formally, we need a couple of definitions.

Definition 1.2.2. *Fix languages $L_1, L_2 \subseteq \{0,1\}^*$. We say that L_1 polynomial-time reduces to L_2 (written $L_1 \leq_p L_2$) if there is a polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that for any $x \in \{0,1\}^*$, $x \in L_1$ iff $f(x) \in L_2$.*

The above definition captures our notion of L_2 being at least as hard to solve as L_1. In particular, note that if L_2 has a polynomial-time algorithm, so does L_1. Now, armed with this definition, we can state what we mean when we say that a problem is as hard as any other problem in NP.

Definition 1.2.3. *A language L is* NP-hard *iff each $L' \in$ NP polynomial-time reduces to L. If L is NP-hard and also happens to belong to NP, then we say that L is* NP-complete*.*

We can now state Cook's theorem.

Theorem 1.2.4 (Cook's Theorem)**.** *Satisfiability is NP-complete.*

Building on Cook's work, Karp, in 1972, showed that a whole range of interesting combinatorial algorithms were all NP-complete also, showing that NP-completeness was a natural phenomenon worthy of further study. These problems included Vertex Cover (the problem of checking if, in a given input graph, there is a small set of vertices that cover all the edges), the Travelling Salesperson problem (checking if there is a small tour of a collection of cities that visits each city exactly once), and Max-Cut (the problem of checking if there is a partition of the vertices of a given graph into two pieces such that many edges go between the pieces).

Theorem 1.2.5 (Karp's Theorem)**.** *Vertex Cover, Travelling Salesperson Problem, and Max-Cut are all NP-complete.*

The framework of NP-completeness can be used to show that many interesting Combinatorial Optimization problems are NP-hard: by this, we mean that the corresponding decision problems are NP-hard.

1.3 Coping with NP-hardness

When an interesting Combinatorial Optimization problem has been proved to be NP-hard, what should one do next? Assuming $\mathrm{P} \neq \mathrm{NP}$, there is unlikely to be an efficient algorithm that solves it completely. How does one then cope with such NP-hardness? Here are two popular approaches:

- **Heuristics**: One could try to come up with an algorithm that worked on "typical" instances of the problem. The problem with this approach is that it is hard to define "typical": there may be algorithms that work on "most" instances, but yet real-life instances may come from a distribution that is mostly supported on the bad instances for this algorithm. Nevertheless, this approach is used commonly and in certain cases, one can demonstrate certain theoretical guarantees. This will however, be beyond the scope of this course.
- **Approximation algorithms**: Here, one scales down one's hopes of finding the optimum solution to the problem, and tries to look for provably "close to optimum" solutions.

  To phrase this more formally, say we are dealing with a Combinatorial Optimization problem associated with the tuple $(\mathcal{X}, \mathcal{Y}, \mathrm{Feas}, \mathrm{Objective})$ where the aim is to maximize the Objective function. Given an $x \in \mathcal{X}$, a $y \in \mathcal{Y}$ is said to be an *α-approximate* solution (for $\alpha \geq 1$) if y is feasible and

  $$\frac{\mathrm{OPT}(x)}{\alpha} \leq \mathrm{Objective}(x, y) \leq \mathrm{OPT}(x)$$

  (For *minimization* problems, a feasible y is α-approximate if $\mathrm{OPT}(x) \leq \mathrm{Objective}(x, y) \leq \alpha\mathrm{OPT}(x)$.)

  An algorithm A is said to be an *α-approximation algorithm* if for each $x \in \mathcal{X}$, the element $A(x) \in \mathcal{Y}$ is an α-approximate solution. The Combinatorial Optimization problem is said to be *α-approximable* if it has a polynomial-time α-approximation algorithm.

Approximation algorithms will be the focus of this course. We give a few examples of approximation algorithms for NP-hard Combinatorial Optimization problems below.

1.3.1 A 2-approximation for Vertex Cover

The Vertex Cover problem is defined as follows:

- INPUT: An undirected graph $G = (V, E)$.
- OUTPUT: Find a cover $C \subseteq V$ such that $\forall (u, v) \in E$, either $u \in C$ or $v \in C$.

- OBJECTIVE: Minimize $|C|$.

We want an efficient algorithm A such that for any graph G, we have

$$\mathrm{OPT}(G) \leq |A(G)| \leq 2\mathrm{OPT}(G)$$

Note the dilemma here: we want to prove that the algorithm outputs a solution that is at most twice as bad as the optimal solution, but we have no handle on the optimal solution itself (at least, we have no algorithm to compute it). This is a hurdle we will need to overcome in each of the problems we consider. The way we will overcome it here is to upper bound $|A(G)|$ by 2 times a quantity that is an efficiently computable *lower bound* on the size of the minimum vertex cover of G.

The lower bound we use is the size of any maximal matching in the graph G. Clearly, the size of any matching in G – and hence, in particular, a maximal matching – is a lower bound on the size of any vertex cover of G. More interestingly, however, the following is true (the simple proof is left to the reader):

Claim 1.3.1. *If M is a maximal matching in G, then the set of endpoints of the edges in M form a vertex cover of G.*

Given the above claim, there is an easy way to get a 2-approximation to Vertex Cover: Find any maximal matching M in G and output all the endpoints of edges that occur in M. The output set is of cardinality at most twice the cardinality of M, which is at most twice the size of the minimum vertex cover.

Is the analysis of the above algorithm optimal in terms of approximation ratios? The answer is easily seen to be yes, since on an input graph that is simply a collection of disjoint edges, the algorithm outputs a set that is twice the size of the minimum vertex cover.

Moreover, one can also see that the lower bound we are using – the size of a maximal matching – can be a factor of 2 less than the size of the minimum vertex cover: an example of a graph where this happens is a clique with an odd number of vertices. This tells us that any analysis that uses the size of a maximal matching as a lower bound on the vertex cover cannot be used to prove that an algorithm is an α-approximation, for any constant $\alpha < 2$.

These explain why the above algorithm can perform no better. We will later see in this course that in fact no other algorithm can perform better assuming some complexity assumptions.

1.3.2 The Travelling Salesperson Problem

The Travelling Salesperson Problem (TSP) is defined as follows:

- INPUT: A distance function $d : [n] \times [n] \to \mathbb{R}^{\geq 0}$ between n vertices of a complete graph.

- OUTPUT: Find a tour $v_0, v_1, \ldots, v_n$ of the vertices s.t. $v_0 = v_n$ and each vertex appears exactly once in the list $\{v_0, v_1, \ldots, v_{n-1}\}$.
- OBJECTIVE: Minimize the total length $\sum_{i=0}^{n-1} d(v_i, v_{i+1})$ of the tour.

The Metric-TSP problem is the special case of this problem where the input distance function is a metric: i.e it is a symmetric function satisfying the triangle inequality:

$$\forall x, y, z \in [n] \quad d(x, z) \leq d(x, y) + d(y, z)$$

It turns out that the general TSP (without assuming the metric property) cannot have an efficient α-approximation algorithm for any α that is polynomial-time computable unless P = NP. This follows from a simple reduction from the well-known NP-complete problem called the Hamiltonian Cycle problem: the input here is a directed graph and we need to check if the graph contains a Hamiltonian cycle, a cycle in which each vertex of the graph appears exactly once.

Claim 1.3.2. *For any $\alpha(n)$ that can be computed in time* poly(n)*, the TSP is not $\alpha(n)$-approximable unless* P = NP*.*

Proof. We prove the claim by showing that if the TSP does have an efficient $\alpha(n)$-approximation algorithm, then we can use such an algorithm to design an algorithm that solves the Hamiltonian cycle problem. Given an input graph $G = ([n], E)$ to the Hamiltonian cycle problem, we construct an instance of the TSP as follows: a distance function is defined on $[n]$ with

$$d(i, j) = \begin{cases} 1, & \text{if } (i, j) \in E, \\ \alpha(n) \cdot n + 1 & \text{otherwise.} \end{cases}$$

If G has a Hamiltonian cycle, then the optimum of the instance of the TSP problem is at most n; otherwise, the optimum is at least $\alpha(n) \cdot n + 1$. If the TSP were $\alpha(n)$-approximable, we would be able to efficiently distinguish between these two cases and hence, we would have a polynomial-time algorithm for the Hamiltonian cycle problem which would imply that P = NP. □

Note that the instance of TSP constructed in the above reduction is *not* an instance of Metric-TSP. Hence, we can still hope for an efficient algorithm to approximate the optimum to this problem. Indeed, we now show that this problem has a 3/2-approximation algorithm.

We start by describing a simpler 2-approximation algorithm for the problem. Let G be the weighted graph corresponding to a given instance of the TSP (the weight of the edge (i, j) is $d(i, j)$). As in the case of vertex cover, we would first like an efficiently computable lower bound to the optimum. This lower bound, in the case of the Metric TSP, is the weight of the Minimum Spanning Tree (MST) of G. To see that this is indeed a lower bound, note that removing any edge from any TSP tour of the vertices gives us a spanning tree of G, the weight of which is of course at least the weight of the MST of G.

Now, we use the MST of G to compute a short tour of all the vertices of G as follows:

- Find an MST T of G.
- Consider the subgraph H of G obtained by adding to T edges (i, j) for each i, j such that $(j, i) \in T$.
- Since the above graph is Eulerian, we can efficiently find an Eulerian tour of H. Output this tour.

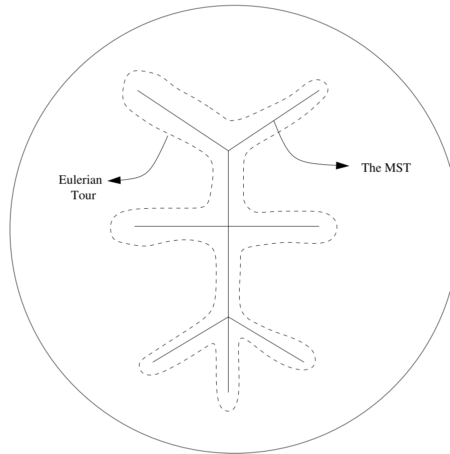


Figure 1: Finding an Eulerian Tour from the MST

Note that the weight of the tour output by the above algorithm is twice the weight of T and hence at most twice the weight of the optimum TSP tour. To obtain a TSP tour of at most the same weight, we use the following useful fact about instances of Metric TSP.

Lemma 1.3.3. *Let (G, d) be an instance of Metric TSP. Given any tour $v_0, v_1, \ldots, v_m$ of all the vertices of G, one can find efficiently a TSP tour $u_0, u_1, \ldots, u_n$ of at most the same weight as the given tour.*

Proof. The proof is simple. We define the TSP tour $u_0, u_1, \ldots, u_n$ to be a subsequence of $v_0, v_1, \ldots, v_m$ as follows. Let $u_0 = v_0$. Say u_i has been defined and is equal to v_j. Then, we define u_{i+1} to be v_k for the least k such that $k > j$ and v_j has not yet been visited in the path $u_0, u_1, \ldots, u_i$; if no such k exists, then we set $u_n = v_m$. It is easy to prove, by the triangle inequality, that each such "short-cut" we take can only be of shorter length than the walk along the v_i's. Hence, the length of the TSP tour obtained is at most the length of the input tour. □

The description of the approximation algorithm A is now complete: we construct the Eulerian tour as above and then obtain a TSP tour from it as described in Lemma 1.3.3. Clearly, we have

$$\mathrm{OPT}(G, d) \leq A(G, d) \leq 2\mathrm{OPT}(G, d)$$

Thus, A is a 2-approximation algorithm for Metric TSP. To obtain a 3/2-approximation algorithm, we notice that we do not *need* to double the edges of T above to get an Eulerian graph and hence an Eulerian tour: any method of obtaining an Eulerian graph from T by adding only a small-weight set of edges to T will do.

Note that since we can reorient the edges of G as we please (since the metric is symmetric), we would be done if the degrees of each vertex in T were even. (Of course, this cannot happen since T is a tree and must have degree-1 vertices.) So the "bad" vertices are the odd-degree vertices. We would like to add a small-weight set of edges to T so that the degree of every vertex is even. Let B be the set of bad vertices. Surely, $|B|$ is even. Hence, a simple way of getting a graph where every vertex has even degree is to add to T a matching that saturates exactly the vertices in B. We add a minimum weight such matching M to T.

We claim that the weight of such a matching is small: it is at most half the weight of the optimum TSP tour.

Claim 1.3.4. *Let M be defined as above. Then,*

$$\sum_{e \in M} d(e) \leq \frac{\mathrm{OPT}(G, d)}{2}$$

Note that, assuming the above claim, we can obtain an Eulerian tour, and hence a TSP tour, of weight at most $3/2\mathrm{OPT(G,d)}$. Thus, we have a 3/2-approximation algorithm.

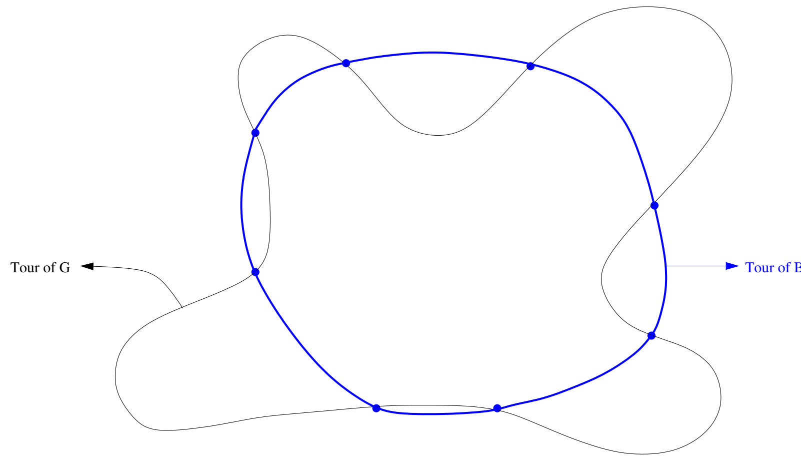


Figure 2: Finding a tour of B from a tour of G

Proof of Claim. Consider any optimal TSP tour $v_0, v_1, \ldots, v_n$ of G. Note that, by removing any vertex v_i from this tour, by the triangle inequality, one can only shorten its length. Thus, by removing vertices not in B one after another, we end up with a TSP tour of B of at most the same weight.

This TSP tour of B is a disjoint union of two matchings M_1 and M_2 saturating B. Hence, at least one of them is at most half the weight of the TSP tour. Since M is a minimum weight matching on the vertices in B, the claim follows. □

This improvement from 2 to 3/2 is due to Christofides [Chr76]. In fact, the recent result on approximation algorithms for asymmetric metric TSP due to Asadpour, Goemans, Mdry, Oveis-Gharan, and Saberi [AGM+10] (which won the best paper award in SODA 2010) uses a similar idea to improve the approximation factor from $O(\log n)$ (due to Frieze, Galbiati and Maffioli [FGM82]) to $O(\log n/\log\log n)$. However, the above algorithm for symmetric metric-TSP due to Christofides [Chr76] remains unimproved since 1976.

References

[AGM+10] Arash Asadpour, Michel X. Goemans, Aleksander Madry, Shayan Oveis-Gharan, and Amin Saberi. *An $O(\log n/\log\log n)$-approximation algorithm for the asymmetric traveling salesman problem*. In *Proc. 21th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2010.

[Chr76] Nicos Christofides. *Worst-case analysis of a new heuristic for the travelling salesman problem.* Technical Report 388, Graduate School of Industrial Administration, Carnegie-Mellon University, 1976.

[FGM82] Alan M. Frieze, Giulia Galbiati, and Francesco Maffioli. *On the worst-case performance of some algorithms for the asymmetric traveling salesman problem.* Networks, 12(1):23–39, 1982. doi:10.1002/net.3230120103.

[HC09] Prahladh Harsha and Moses Charikar. *Limits of approximation algorithms: PCPs and unique games*, 2009. (DIMACS Tutorial, July 20-21, 2009). arXiv:1002.3864.

[Sud99] Madhu Sudan. *6.893: Approximability of optimization problems*, 1999. (A course on Approximability of Optimization Problems at MIT, Fall 1999).

[Vaz04] Vijay V. Vazirani. *Approximation Algorithms.* Springer, 2004.