Algorithm Design XVIII

Coping with NP-Completeness I: Approximation Algorithm

Guoqiang Li
School of Software

SHANGHAI JIAO TONG UNIVERSITY

Coping with NP-completeness

Q. Suppose I need to solve an **NP**-complete problem. What should I do?

A. Theory says you're unlikely to find polynomial time algorithm.

Must sacrifice one of three desired features.

- Solve problem to optimality.
- Solve problem in polynomial time.
- Solve arbitrary instances of the problem.

Think About: What features have been sacrificed in today's topic?

Approximation Algorithms

Optimization Problems

In an optimization problem we are given an instance I and are asked to find the optimum solution:

- The one with the maximum gain if we have a maximization problem like INDEPENDENT SET,or
- The minimum cost if we are dealing with a minimization problem such as the TSP.

For every instance I , $OPT(I)$ denotes the value (benefit or cost) of the optimum solution.

We always assume $OPT(I)$ is a positive integer (we may write as OPT when context is clear).

Approximation Ratio

We have seen the greedy algorithm for SET COVER:

For any instance I of size n, this greedy algorithm quickly finds a set cover of cardinality at most $OPT(I) \cdot \log n$.

This $\log n$ factor is known as the approximation guarantee of the algorithm.

Suppose now that we have an algorithm $\mathcal{A}$ for a minimization problem which, given an instance I, returns a solution with value $\mathcal{A}(I)$.

The approximation guarantee of $\mathcal{A}$ is defined to be

$$\alpha_{\mathcal{A}} = \max_{I} \frac{\mathcal{A}(I)}{OPT(I)}$$

A Dilemma

To establish the approximation guarantee, the cost of the solution produced by the algorithm needs to compare with an optimal solution.

For such problems, not only is it NP-hard to find an optimal solution, but it is also NP-hard to compute the cost of an optimal solution.

In fact, computing the cost of an optimal solution is precisely the difficult core of such problems.

How do we establish the approximation guarantee? The answer provides a key step in the design of approximation algorithms.

Vertex Cover

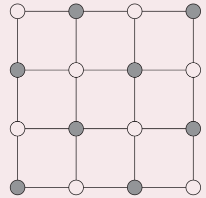
Vertex cover

Vertex cover

Input: An undirected graph $G = (V, E)$.

Output: A subset of the vertices $S \subseteq V$ that touches every edge.

Goal: Minimize $|S|$.

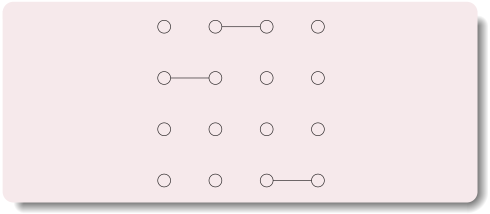


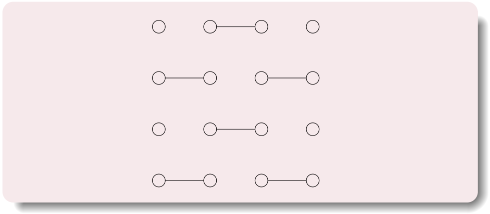
Matching

Given a graph $G = (V, E)$, a subset of the edges $M \subseteq E$ is said to be a matching if no two edges of M share an endpoint.

A matching of maximum cardinality in G is called a maximum matching.

A matching that is maximal under inclusion is called a maximal matching.





Matching

Given a graph $G = (V, E)$, a subset of the edges $M \subseteq E$ is said to be a matching if no two edges of M share an endpoint.

A matching of maximum cardinality in G is called a maximum matching.

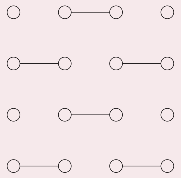
A matching that is maximal under inclusion is called a maximal matching.

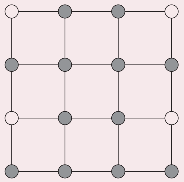
A maximal matching can clearly be computed in polynomial time by simply greedily picking edges and removing endpoints of picked edges. More sophisticated means lead to polynomial time algorithms for finding a maximum matching as well.

Approximation for VERTEX COVER

Algorithm

Find a maximal matching in G and output the set of matched vertices.





Approximation Guarantee Factor

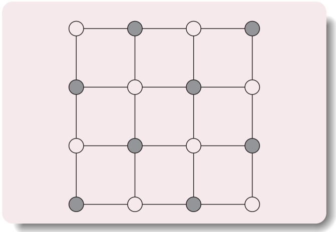
The Algorithm is a factor 2 approximation algorithm for the vertex cover problem.

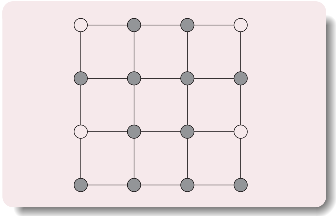
Proof.

- No edge can be left uncovered by the set of vertices picked.
- Let M be the matching picked. As argued above,

$$|M| \leq OPT$$

- The approximation factor is at most $2 \cdot OPT$.





Lower Bounding OPT

The approximation algorithm for vertex cover was very much related to, and followed naturally from, the lower bounding scheme. This is in fact typical in the design of approximation algorithms.

Can the Guarantee be Improved?

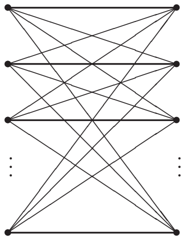
Can the approximation guarantee of Algorithm be improved by a better analysis?

Can an approximation algorithm with a better guarantee be designed using the lower bounding scheme of Algorithm?

Is there some other lower bounding method that can lead to an improved approximation guarantee for VERTEX COVER?

A Better Analysis?

Consider the infinite family of instances given by the complete bipartite graphs $K_{n,n}$.



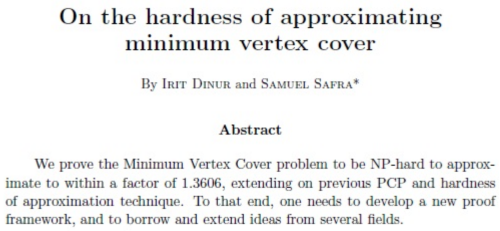
When run on $K_{n,n}$, Algorithm will pick all $2n$ vertices, whereas picking one side of the bipartition gives a cover of size n.

A Better Guarantee?

The lower bound, of size of a maximal matching, is half the size of an optimal vertex cover for the following infinite family of instances. Consider the complete graph K_n, where n is odd. The size of any maximal matching is $(n-1)/2$, whereas the size of an optimal cover is $n-1$.

A Better Algorithm?

Still Open!


On the hardness of approximating minimum vertex cover

By IRIT DINUR and SAMUEL SAFRA*

Abstract

We prove the Minimum Vertex Cover problem to be NP-hard to approximate to within a factor of 1.3606, extending on previous PCP and hardness of approximation technique. To that end, one needs to develop a new proof framework, and to borrow and extend ideas from several fields.

Clustering

Euclidean Distance $d(x, y)$

1. $d(x, y) \geq 0$ for all x, y.
2. $d(x, y) = 0$ if and only if $x = y$.
3. $d(x, y) = d(y, x)$.
4. (Triangle inequality) $d(x, y) \leq d(x, z) + d(z, y)$.

k-Cluster

Definition (k-Cluster)

Input: Points $X = \{x_1, \ldots, x_n\}$ with underlying distance metric $d(\cdot, \cdot)$, integer k.

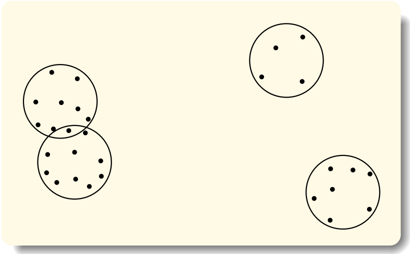
Output: A partition of the points into k clusters $C_1, \ldots, C_k$.

Goal: Minimize the diameter of the clusters,

$$\min_j \max_{x_a, x_b \in C_j} d(x_a, x_b)$$

Theorem

k-clustering is NP-hard.



k-Cluster

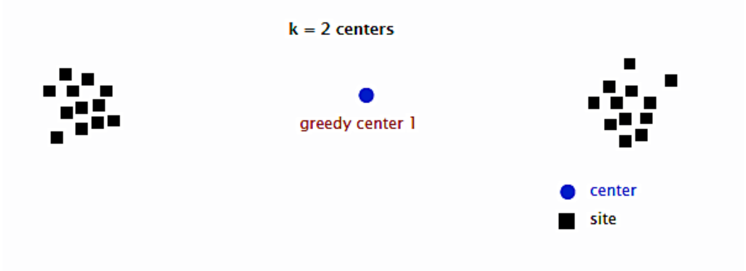
Remark

Search can be infinite!

Greedy algorithm. Put the first center at the best possible location for a single center, and then keep adding centers so as to reduce the covering radius each time by as much as possible.

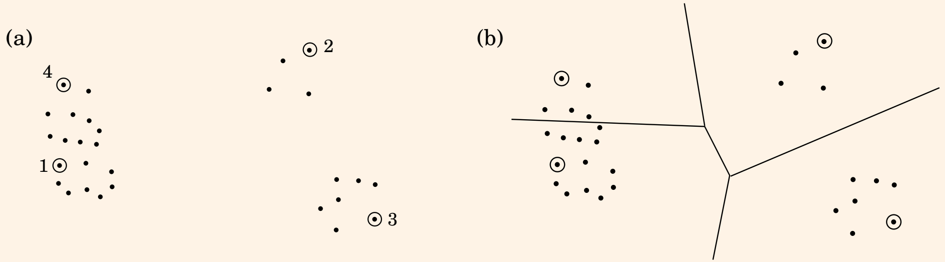
Remark

Greedy algorithm is arbitrarily bad!



Approximation Algorithm

```
Pick any point $\mu_1 \in X$ as the first cluster center
for $i = 2$ to $k$:
  Let $\mu_i$ be the point in $X$ that is farthest from $\mu_1, \ldots, \mu_{i-1}$
  (i.e., that maximizes $\min_{j<i} d(\cdot, \mu_j)$)
Create $k$ clusters:  $C_i = \{$all $x \in X$ whose closest center is $\mu_i\}$
```



Proof for the Approximation Ratio 2

Let $x \in X$ be the point farthest from $\mu_1, \ldots, \mu_k$, and r be its distance to its closest center.

Then every point in X must be within distance r of its cluster center. By the triangle inequality, this means that every cluster has diameter at most $2r$.

We have identified $k+1$ points $\{\mu_1, \mu_2, \ldots, \mu_k, x\}$ that are all at a distance at least r from each other.

Any partition into k clusters must put two of these points in the same cluster and must therefore have diameter at least r.

Remark

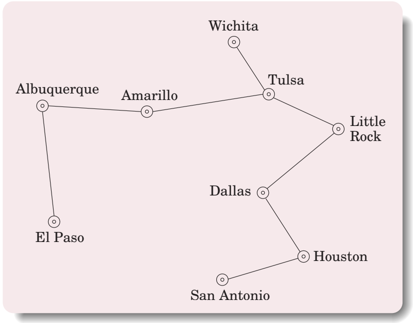
No better approximation algorithm for this problem so far.

TSP

TSP on Metric Space

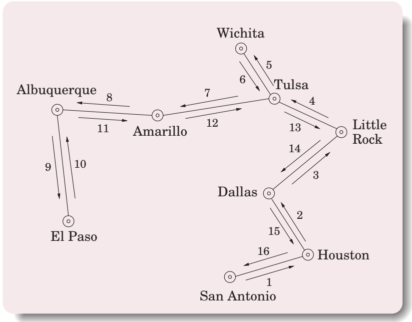
Removing any edge from a traveling salesman tour leaves a path through all the vertices, which is a spanning tree.

Therefore, TSP cost $\geq$ cost of this path $\geq$ MST cost



TSP on Metric Space

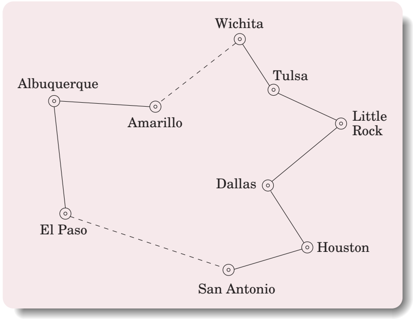
If we can use each edge twice, then by following the shape of the MST we end up with a tour that visits all the cities, some of them more than once.



TSP on Metric Space

To fix the problem, the tour should simply skip any city it is about to revisit, and instead move directly to the next new city in its list.

By the triangle inequality, these bypasses can only make the overall tour shorter.



A Simple Factor 2 Algorithm

Consider the following algorithm:

1. Find an MST, T of G
2. Double every edge of the MST to obtain an Eulerian graph.
3. Find an Eulerian tour, $\mathcal{T}$, on this graph.
4. Output the tour that visits vertices of G in the order of their first appearance in $\mathcal{T}$. Let $\mathcal{C}$ be this tour.

The above algorithm is a factor 2 approximation algorithm for metric TSP.

Analysis

SHANGHAI JIAO TONG UNIVERSITY

$cost(T) \leq$OPT.

Since $\mathcal{T}$ contains each edge of T twice, $\mathrm{cost}(\mathcal{T}) = 2 \cdot \mathrm{cost}(T)$.

Because of triangle inequality, after the short-cutting (step 4) step, $\mathrm{cost}(\mathcal{C}) \leq \mathrm{cost}(\mathcal{T})$.

Combining these inequalities we get that

$$\mathrm{cost}(\mathcal{C}) \leq 2 \cdot \mathrm{OPT}.$$

General TSP

What if we are interested in instances of TSP that do not satisfy the triangle inequality?

It turns out that this is a much harder problem to approximate.

Recall we gave a polynomial-time reduction which given any graph G and integer any $C > 0$ produces an instance $I(G, C)$ of the TSP such that:

1. If G has a Rudrata cycle then $OPT(I(G, C)) = n$, the number of vertices in G.
2. If G has no Rudrata cycle, then $OPT(I(G, C)) \geq n + C$.

This means that even an approximate solution to TSP would enable us to solve RUDRATA CYCLE.

General TSP

Consider an approximation algorithm $\mathcal{A}$ for TSP and let $\alpha_{\mathcal{A}}$ denote its approximation ratio.

From any instance G of RUDRATA CYCLE, we will create an instance $I(G,C)$ of TSP using the specific constant

$$C = n \cdot \alpha_{\mathcal{A}}$$

```
Given any graph $G$:
   compute $I(G,C)$ (with $C = n \cdot \alpha_{\mathcal{A}}$) and run algorithm $\mathcal{A}$ on it
   if the resulting tour has length $\leq n\alpha_{\mathcal{A}}$:
      conclude that $G$ has a Rudrata path
   else:  conclude that $G$ has no Rudrata path
```

Knapsack

Knapsack

Recall the Knapsack problem: There are n items, with weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ (all positive integers), and the goal is to pick the most valuable combination of items subject to the constraint that their total weight is at most W.

Earlier we saw a dynamic programming solution to this problem with running time $O(nW)$. Using a similar technique, a running time of $O(nV)$ can also be achieved, where V is the sum of the values.

Neither of these running times is polynomial, because W and V can be very large, exponential in the size of the input.

Knapsack

Let's consider the $O(nV)$ algorithm.

In the bad case when V is large, what if we simply scale down all the values in some way?

For instance, if

$$v_1 = 117,586,003, \qquad v_2 = 738,493,291, \qquad v_3 = 238,827,453$$

we could simply knock off some precision and instead use 117, 738, and 238.

This doesn't change the problem all that much and will make the algorithm much, much faster!

Knapsack

Along with the input, the user is assumed to have specified some approximation factor $\epsilon > 0$.

```
Discard any item with weight $> W$
Let $v_{\max} = \max_i v_i$
Rescale values $\widehat{v}_i = \lfloor v_i \cdot \frac{n}{\epsilon v_{\max}} \rfloor$
Run the dynamic programming algorithm with values $\{\widehat{v}_i\}$
Output the resulting choice of items
```

Since the rescaled values $\hat{v}_i$ are all at most n/ϵ, the dynamic program is efficient, running in time

$$O(n^3/\epsilon)$$

Knapsack

Suppose the optimal solution to the original problem is to pick some subset of items S, with total value K^*.

The rescaled value of this same assignment is

$$\sum_{i \in S} \hat{v}_i = \sum_{i \in S} \lfloor v_i \cdot \frac{n}{\epsilon \cdot v_{max}} \rfloor \geq \sum_{i \in S} (v_i \cdot \frac{n}{\epsilon \cdot v_{max}} - 1) \geq K^* \cdot \frac{n}{\epsilon \cdot v_{max}} - n$$

Therefore, the optimal assignment for the shrunken problem, call it $\hat{S}$, has a rescaled value of at least this much.

In terms of the original values, assignment $\hat{S}$ has a value of at least

$$\sum_{i \in \hat{S}} v_i \geq \sum_{i \in \hat{S}} \hat{v}_i \cdot \frac{\epsilon \cdot v_{max}}{n} \geq (K^* \cdot \frac{n}{\epsilon \cdot v_{max}} - n) \cdot \frac{\epsilon \cdot v_{max}}{n} = K^* - \epsilon \cdot v_{max} \geq K^*(1 - \epsilon)$$

The Approximability Hierarchy

The Approximability Hierarchy

All NP-complete optimization problems are classified as follows:

- Those for which, like the TSP, no finite approximation ratio is possible.
- Those for which an approximation ratio is possible, but there are limits to how small this can be: VERTEX COVER, k-CLUSTER, and the TSP with triangle inequality.
- Down below we have a more fortunate class of NP-complete problems for which approximability has no limits, and polynomial approximation algorithms with error ratios arbitrarily close to zero exist: KNAPSACK.
- Finally, there is another class of problems, between the first two given here, for which the approximation ratio is about $\log n$: SET COVER.