

Last update: 16/02/2006, 18:58

31/10/05

Travelling Salesman Problem

Input: Cities $C = \{c_1, \dots, c_n\}$, Distances d_{ij}

Definition: A Tour is a permutation π on $\{1, \dots, n\}$

Decision Problem: Is there a tour of length $\leq B$ (for a given B)

TSP_{Bound} : What is the length of the shortest tour, B^* ?

TSP_{opt} : Find the shortest tour, Θ^*

TSP_{all} : Find ALL the shortest tours

Polynomial Transformation

$f : \Sigma_1^* \rightarrow \Sigma_2^*$ is a poly. trans. from L_1 to L_2 if:

1. f can be calculated by a poly. det. algorithm.
2. $\forall x \in \Sigma_1^*, x \in L_1 \Leftrightarrow f(x) \in L_2$

If this f exists, then $L_1 \propto L_2$

L is NP-Complete if:

1. $L \in NP$
2. $\forall L' \in NP, L' \propto L$

Polynomial Reduction

$L_1 \propto_T L_2$ if given procedure P_2 for solving L_2 , there exists a poly. alg. A which solves L_1 by using poly. many calls to P_2

L is NP-Hard if $\forall L' \in NP, L' \propto_T L$

K-Largest-Subset

Input: $\bar{C} = \{c_1, \dots, c_n\}, K, B$

Output: Are there at least K different subsets of \bar{C} , summing to $\leq B$

KLS is NP-Hard

Partition

Input: $\bar{C} = \{c_1, \dots, c_n\}$

Output: Can \bar{C} be split into $\bar{C} = A \cup B, A \cap B = \emptyset$ s.t. $\sum_{c_i \in B} c_i = \sum_{c_i \in A} c_i$

Partition \propto_T KLS

NP-Easy

L is NP-Easy if $\exists L' \in NP$ s.t. $L \propto_T L'$

Self-Reducibility

A problem Π is self-reducible if its Optimization (Calculation) version can be solved using its Decision version (in poly. time).

Example: TSP

7/11/05

Self-Reducibility cont.

Clique

Input: Graph G , Natural K

Output: Does G have a clique of size $\geq K$?

Opt version: Find the largest clique in G .

Clique is self-reducible

Dominating Set

Input: G, d

Def: W is dominating in G if $\forall v \in G$, either $v \in W$ or $v \in \Gamma(W)$

Output: Does G have a dominating set of size $\leq K$?

DS is Self-reducible

Euc-TSP

TSP, in which $c_i = \{x_i, y_i\}$, $d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$

Also a self-reducible problem

Pseudo-Poly. Algorithms

Definition: For an input of numbers: $I = \langle \alpha_1, \dots, \alpha_n \rangle$, define $Max(I) = \max_{1 \leq i \leq n} \{\alpha_i\}$. An algorithm is Pseudo-Polynomialic if \exists poly. $p(x, y)$ s.t. for every input I , $T_A(I) \leq p(length(I), max(I))$

(Note: if $max(I) \leq q(length(I))$ then a pseudo-poly. alg. is polynomialic)

Knapsack

A subset of the knapsack problem:

Input: Sizes $\bar{S} = \{s_1, \dots, s_n\}$, Size limit B

Output: Are there x_1, \dots, x_n s.t. $\sum x_i s_i = B$

Pseudo-poly. algo. for it: Build a graph $G_{\bar{S}, B} = (V, E)$; $V = \{0, 1, \dots, B\}$; $E = \{(i, j) : \exists s \in \bar{S}, i + s = j\}$. Find a directed route from 1 to B (add neighbours of nodes in the set until done or reached B).

The algorithm has time $O(n \cdot B)$ which is pseudo-poly.

Strongly NP-Hard

Let π be an NP-Hard problem. For a poly. q , let π_q be the restriction of π to inputs I s.t. $max(I) \leq q(length(I))$.

π is called Strongly NP-hard if π_q is NP-hard

Examples: Clique, TSP

Pseudo-Poly Alg. For 0-1 Knapsack

In 0-1 knapsack, $x_i \in \{0, 1\}$

Dynamic Programming Alg.:

We shall fill a table $T_{n \times (B+1)}$ s.t. $T(i, b) = \begin{cases} 1 & \exists \text{ subset of } \{s_1, \dots, s_i\} \text{ summing to } b \\ 0 & \text{o.w.} \end{cases}$

The requested answer is then $T(n, B)$. Filling rules:

R1 (For the first row): $T(1, b) = \begin{cases} 1 & b = 0 \vee b = s_1 \\ 0 & \text{o.w.} \end{cases}$

R2 (other rows): $T(i+1, b) = \begin{cases} 1 & T(i, b) = 1 \vee T(i, b - s_{i+1}) = 1 \\ 0 & \text{o.w.} \end{cases}$

Total complexity: $O(nB)$

Exp. Algo. for TSP

Dynamic Programming Alg.:

Define: $S \subseteq M, M = \{2, \dots, n\}, F(S, i) = \text{length of the shortest route starting at 1, ending at } i \text{ and visits every city in } S \text{ exactly once (for } i \in S)$

Filling F (in a growing order of $|S|$):

R1 (for $|S| = 1$): $S = \{i\} : F(\{i\}, i) = d_{1i}$

R2: $F(S, i) = \min_{j \in S, j \neq i} \{F(S - \{i\}, j) + d_{ji}\}$

The final answer: $\min_{2 \leq i \leq n} \{F(M, i) + d_{i1}\}$

Complexity: less than $n \cdot n \cdot 2^n$ (naive algo. takes $n! \approx n^n$)

14/11/05

Exact 3-SAT

Input: Formula φ in CNF form over $\{x_1, \dots, x_n\}$

Output: Is there an assignment which satisfies exactly one literal in each clause.

Definition: A variable is a singleton if it appears in a single clause.

Definition - Canonical form:

1. Each two clauses have at most one common variable
2. Each clause contains at most one singleton
3. Each clause contains exactly 3 different variables

Bringing a formula to a canonical (or empty) form is polynomial.

Procedure simple (solve when each clause has a singleton): Change each $C = (x_1, u_2, u_3)$ with $\hat{C} = \bar{u}_2 \vee \bar{u}_3$, and solve as a 2SAT formula (linear time)

Algorithm X3SAT (on a canonical φ):

1. If each clause has a singleton, call $\text{simple}(\varphi)$ and finish
2. (Else)
 - (a) If $\exists x_i$ that appears both positive and negative, choose it
 - (b) O.w., choose an x_i from a clause which has no singletons
3. $b \leftarrow \text{TEST}(\varphi, x_i, T)$
4. If $b = T$, return T and finish
5. Else, $b \leftarrow \text{TEST}(\varphi, x_i, F)$, and return b

Procedure TEST(φ, x_i, v):

1. Set $\tau(x_i) \leftarrow v$
2. Simplify φ to a canonical form φ'
 - If found a contradiction, return F
 - If $\varphi' = \emptyset$, return T
3. Return X3SAT(φ')

Complexity: $2^{\alpha n}$ for some $\alpha < 1$ (we analyzed one case, in which $\alpha = \frac{1}{4}$)

MIS - Maximum Independent Set

An independent set in a graph G is a set of vertices s.t. there's no edges between them.

Maximal Independent set - no vertices can be added to it.

Maximum Independent Set - The biggest possible IS in the graph.

MIS has a recursive algo. with complexity $\approx 2^{0.32n}$

MIS in a planar graph

The Small Separator THM: Given a planar graph G with n vertices, \exists vertices $S \subseteq V$. Erasing S cuts the graph into A_1, A_2 s.t.:

1. $|S| \leq \sqrt{8n}$
2. $|A_1, A_2| \leq \frac{2}{3}n$
3. There are no edges between A_1 and A_2

This S can be found in a poly. time.

Recursive Algo. For MIS in a planar graph:

1. Find S, A_1, A_2 from the THM.
2. For each independent subset $M_0 \subseteq S$, do:
 - (a) Remove from A_1, A_2 neighbours of M_0 , and get A'_1, A'_2
 - (b) $M_1 \leftarrow MIS(G(A'_1))$
 - (c) $M_2 \leftarrow MIS(G(A'_2))$
 - (d) $M \leftarrow M_0 \cup M_1 \cup M_2$
3. Return the biggest M found.

This algorithm finds an MIS, and has time complexity $2^{c\sqrt{n}}$ for some c

21/11/05

0-1 Knapsack

The naive algorithm for 0-1 knapsack takes $O(n \cdot 2^n)$

Algo. with $T = O(n \cdot 2^{n/2})$

1. Define $A_\alpha = \{A_1, \dots, A_{n/2}\}$
2. Define $A_\beta = \{A_{n/2+1}, \dots, A_n\}$
3. Create an increasing list α of all subsets of A_α , and an increasing list β of all subsets of A_β
4. Initialize $I \leftarrow 1, J \leftarrow 2^{n/2}$
5. while $I \leq 2^{n/2}$ and $J \geq 1$
 - If $\alpha_I + \beta_J = B$, halt and return “YES”
 - If $\alpha_I + \beta_J < B, I \leftarrow I + 1$
 - If $\alpha_I + \beta_J > B, J \leftarrow J - 1$

Note: (not 100% sure) If α, β are built sorted (each subset is inserted in the correct place), the algorithm is even $O(2^{n/2})$

A simplified version:

1. Define A_1, A_2 as before
2. Create α, β but sort only α
3. $\forall 1 \leq j \leq 2^{n/2}$, do: binary search in α for $B - \beta_j$

Less memory, more time

1. Define $A_1 = \{a_1, \dots, a_{n/k}\}$, $A_2 = \{a_{n/k+1}, \dots, a_n\}$
2. Create α, β but sort only α
3. $\forall 1 \leq j \leq 2^{\frac{k-1}{k}n}$, do: binary search in α for $B - \beta_j$

Time complexity: $O(n2^{\frac{k-1}{k}n})$

Space complexity: $O(2^{n/k})$

Time \cdot Space = $O(n2^n)$

Using K tables

- Define $A_1 = \{a_1, \dots, a_{n/k}\}, \dots, A_k = \{a_{n-(k-1)n/k+1}, \dots, a_n\}$
- Create sorted lists of subsets R_1, \dots, R_k .
- For each $\beta_i \in R_k$, recursively check for a solution of size $B - \beta_j$ in R_1, \dots, R_{k-1}

Time complexity: $O(2^{\frac{k-1}{k}n})$

Space complexity: $O(k2^{n/k})$

Algo. with $T = O(2^{n/2}), S = O(2^{n/4})$

- Create $\alpha_1 = \{a_1, \dots, a_{n/4}\}, \alpha_2 = \{a_{n/4+1}, \dots, a_{n/2}\}$
- Create a matrix of size $2^{n/4} \times 2^{n/4}$. This matrix will represent sums of subsets from α_1 and α_2

The idea is not to fill the whole matrix, but rather hold only $2^{n/4}$ sums at each given moment. These sums will be held in a priority queue such that the minimum can be found easily, and a new sum can be inserted easily. At each step, check the smallest sum (with the largest sum in B's queue). If needed, remove it and insert the one on its right in the matrix.

28/11/05

Vertex cover in bipartite graphs

Add source and target nodes. Connect one side to the source and the other to the target, and find a maximum flow (=min-cut) (polynomially). Vertices touching cut edges are the cover.

Disconnecting n from k in the complexity

Vertex Cover

The problem: Is there a VC of size k ? The naive algorithm needs time $O(n^k |E|) = O(n^{k+2})$

We would like to get a complexity of $O(n^{c_1} c_2^k)$

Algorithm (Procedure $p(G, k)$):

1. If $E(G) = \emptyset$, return 1. Else, if $k = 0$, return 0.
2. Choose an edge $e = (u, w)$ which isn't covered.
3. $G_u \leftarrow G - E_u$ ($E_u =$ all edges touching vertex u)
4. $b_u \leftarrow p(G_u, k - 1)$
5. $G_w \leftarrow G - E_w$
6. $b_w \leftarrow p(G_w, k - 1)$
7. return $b_u \vee b_w$

This algorithm has time complexity of $O(|E|2^k) = O(n^2 2^k)$

Note that for $k = O(\log n)$, the algorithm is polynomial

Clique

Claim: If the clique problem is poly. for $k = O(\log n)$ then clique is solvable (for any k) in time $2^{\sqrt{n} \log n}$

Proof by building $\tilde{G} = (\tilde{V}, \tilde{E})$ where $\tilde{V} =$ all subsets of V of size \sqrt{k} , and in \tilde{E} there's an edge between each two disjoint sets whose union is a clique in G , and using the assumed-to-exist procedure.

Simple route

Input: Graph g , Natural number k

Output: Is there a simple (acyclic) route in G of length at least k ?

Naive algorithm will take $O(n^k k!)$ time

Algorithm:

- Repeat $e^k \log n$ times:
 - Randomly color the graph in k colors, and look for a “colorful” route of length k

Note: Failure probability is $(1 - \frac{1}{e^k})^{e^k \log n} \approx \frac{1}{n}$

Finding a colorful route (dynamic programming):

Fill a boolean table $T(S, V, C)$ where S, V are source and destination and C is a set of colors of size i , s.t. $T(S, V, C) = 1$ iff \exists a route of length i from S to V , colored exactly by the colors of C . Filling the table:

$$i = 1 \Rightarrow C = \{c\} \Rightarrow T(S, V, C) = \begin{cases} 1 & V = S, \text{ colored } c \\ 0 & \text{o.w.} \end{cases}$$

$$i = 2 \Rightarrow C = \{c_1, c_2\} \Rightarrow T(S, V, C) = \begin{cases} 1 & (S, V) \in E, S \text{ col'd } c_1, V \text{ col'd } c_2 \text{ or the opposite} \\ 0 & \text{o.w.} \end{cases}$$

$i + 1$: Go through the i th level. For each '1' cell, go over the nbrs of w . For each nbr, if its color does not appear in C , mark $T(S, V, C \cup \{c\}) = 1$

If there's a '1' in the last row, a colorful route has been found.

5/12/05

Approximation Algorithms

For an input I , $f^*(I)$ is the best solution. $f_A(I)$ is the solution found by A . Algorithm A is said to have an approximation factor of ϵ if $\forall I, \frac{|f_A(I) - f^*(I)|}{f^*(I)} \leq \epsilon$

($\epsilon \geq 0$)

An alternative definition for minimization problems can be $\frac{f_A(I)}{f^*(I)} \leq \epsilon$ ($\epsilon \geq 1$), and for maximization problems $\frac{f^*(I)}{f_A(I)} \leq \epsilon$

TSP-bound

Claim: If TSP-bound has an approximation algorithm with a constant factor ρ , then $N=NP$ (this claim holds for any $\rho < exp(n)$).

The claim is proven by reducing Hamiltonian-Cycle to TSP-bound, and building a graph G with distances $d_{ij} = \begin{cases} 1 & (i, j) \in E \\ 0 & \text{o.w.} \end{cases}$

Δ TSP - Nearest Neighbour

in Δ TSP distances conform with the triangle inequality.

In the NN heuristics, at each step we have a route. If the route ends at u , the nearest nbr of u which is not in the route is added.

This algo. gives approximation factor of $\epsilon \approx \log n$

Δ TSP - Global Nearest Neighbour

At each step, hold a cycle. Add the node which is closest to one of the cities in the cycle.

This algo. gives an approx. factor of 2 (the solution is at most twice the best solution).

Δ TSP - The Spanning Tree Algorithm

1. Find a Min. Spanning Tree T on G
2. Find a DFS tour Θ' on the tree (each edge is walked twice)
3. Create Θ from Θ' by taking "shortcuts"

This algo. also gives an approx. factor of 2 (this is tight).

Δ TSP - The Euler Cycle Algorithm

1. Find a MST T on G
2. $Z = \{\text{Vertices in } G \text{ with an odd degree}\}$
3. Build a minimum matching M on Z
4. Find an Euler cycle Θ' on $T \cup M$
5. Create Θ from Θ' by taking "shortcuts"

This algo. gives an approx. factor of $3/2$ (this is tight).

19/12/05

Apporximating MIS

Greedy Algorithm

1. $S \leftarrow \emptyset$
2. While $V \neq \emptyset$, do
 - (a) Let v be the vertex with the lowest degree in V
 - (b) $S \leftarrow S \cup \{v\}$
 - (c) $V \leftarrow V - (\{v\} \cup \Gamma(v))$

Lemma 1: If a graph G has maximum degree of Δ , then the greedy algo. gives approx. factor of $\leq \Delta + 1$

Lemma 2: If a graph G is k -colorable, then the greedy algo. finds an independent set of size $\geq \frac{1}{2} \log_k n$

Algorithm with approx. factor of $\frac{n}{\log n}$

1. Cut V into $m = \frac{n}{\log n}$ sets V_1, \dots, V_m of size $|V_i| = \log n$
2. On each V_i exhaustively find its MIS, m_i
3. Return $\max\{m_i\}$

Algorithm with approx. factor of $\approx \frac{n}{\log^2 n}$

Let $|S^*| = \frac{n}{K}$ (we can polynomially check all K 's, so assume K is known)

Consider 3 cases:

1. $K > \log^2 n$: Select a single vertex

2. $\frac{\log n}{c \log \log n} \leq K \leq \log^2 n$:

Algorithm A:

- (a) Define $t = \frac{\log n}{\log \log n}$
- (b) Cut V into $m = \frac{n}{kt}$ sets V_1, \dots, V_m s.t. $|V_i| = kt$
- (c) For each V_i , exhaustively look for an IS of size t
- (d) As soon as you find one, return it.

Algorithm A gives approx factor of $\rho = \frac{n \cdot c \log^2 \log n}{\log^2 n}$

3. $K \leq \frac{\log n}{c \log \log n}$:

Algorithm $B(G, K)$ (For G with no cliques of size K) :

- (a) If $K = 2$, return V
- (b) ($K \geq 3$)
 - i. If $\Delta < n^{1-\frac{1}{k-1}}$, activate the greedy algo.
 - ii. Otherwise, choose a vertex v with a maximal degree and call $S_{k-1} \leftarrow B(\Gamma(v), k-1)$. Return $S_k = S_{k-1}$

Algorithm $Free_k(G)$:

While G has a clique of size k , remove all its vertices

Algorithm C:

(a) $G' \leftarrow Free_K(G)$

(b) Call $B(G, K)$

Since K is not a constant, $Free_k$ is exponential. Therefore, we shall use a version of Ramsie's THM: For a graph G with n vertices, $\forall s, t$ s.t. $\binom{s+t-2}{s-1} \leq n$, G has a clique of size s or an IS of size t . They can be found in poly time.

We shall use this procedure, instead of $Free_k$, with $s = k = 2K$,
 $t = \log^2 n$

9/1/06

PTAS - Polynomial-Time Approximation Schemes

A problem Π has a PTAS if $\forall \epsilon > 0$, \exists an algo A_ϵ that approximates Π with a factor ϵ , and has complexity $p_\epsilon(n)$ (for example $n^{3+1/\epsilon}$)

The algorithm is an FPTAS (fully-PTAS) if the complexity is $p(n, \frac{1}{\epsilon})$ (for example $n^3 (\frac{1}{\epsilon})^2$)

Euc-TSP - no FPTAS

Claim: If $P \neq NP$ then Euc-TSP has no FPTAS (it has a PTAS)

Proof by reduction to HCEG (Hamiltonian Cycle on Euclidean Grid) with $\epsilon = \frac{1}{3n}$

Uniform 0-1 Knapsack - approx. with factor 1/2

Input: Naturals a_1, \dots, a_n , and a natural bound B

Question: Find $S \subseteq \{1, \dots, n\}$ s.t. $\sum_{i \in S} a_i$ is maximal, under the constraint

that $\sum_{i \in S} a_i \leq B$

Algorithm for approx. 1/2: Sort the elements in a non-increasing order of a_i :
 $a_1 \geq a_2 \geq \dots \geq a_n$. Now take elements into the knapsack while you can.

0-1 Knapsack - approx. with factor 1/2

Input: Sizes a_1, \dots, a_n , gains p_1, \dots, p_n , bound B

Question: Find $S \subseteq \{1, \dots, n\}$ s.t. $\sum_{i \in S} p_i$ is maximal under the constraint that $\sum_{i \in S} a_i \leq B$ (or equivalently find $x_1, \dots, x_n \in \{0, 1\}$ s.t. $\sum_i x_i p_i$ is maximal and $\sum_i x_i a_i \leq B$)

Algorithm A:

1. Define $r_i = \frac{p_i}{a_i}$, and sort the items in a non-increasing order of r_i
2. Take items according to this order while you can ($\sum_{i=1}^J a_i \leq B, \sum_{i=1}^{J+1} a_i > B$)
3. Define $S_0 = \{1, \dots, J\}, p_0 = \sum_i^J p_i$
Let i_{max} be the item with the maximal p_i , and define $S_{max} = \{i_{max}\}$
4. If $p_0 > p_{max}$ return S_0 . Otherwise, return S_{max}

Complexity: This implementation takes $O(n \log n)$. Can be optimized by finding a median. If the lower part fits, take it and continue with the upper part. Otherwise, remove the upper part and continue with the lower part.
 $f(n) \leq O(n) + f(n/2)$

FPTAS for 0-1 knapsack

- Exact (exponential) algorithm: Represent a solution by (S, P, A) , $P = \sum_{i \in S} p_i$, $A = \sum_{i \in S} a_i$. The algorithm keeps a list L of solutions.
Initialize: $L = \{(\emptyset, 0, 0)\}$
After iteration 1: $L = \{(\emptyset, 0, 0), (\{1\}, p_1, a_1)\}$
Iteration i : $L = L \cup L'$, $L' = \{(S \cup \{i\}, P + p_i, A + a_i) \mid (S, P, A) \in L\}$

$L, A + a_i \leq B$ (i.e., add to the list all options of adding i to a solution without violating the bound)

- Domination-based Optimization: (S, P, A) dominates (S', P', A') if $P \geq P', A \leq A'$. If so, we can remove (S', P', A') from the list.

L is kept in a decreasing order s.t. $P_1 > P_2 > \dots$ and $A_1 > A_2 > \dots$

This ensures that $|L| \leq B, |L| \leq p^*$

This algorithm takes time $O(n^2 p^*)$

- Approximation using scaling: Define $q_i \leftarrow \lfloor \frac{p_i}{k} \rfloor$, and run the exact algorithm using the q_i 's instead of the p_i 's

In order to maintain an approx factor of ϵ , we need $k \leq \frac{\epsilon p^*}{|S^*|}$

Since we don't know $p^*, |S^*|$, we can use $p_{max} \leq p^*$ and $n \geq |S^*| \Rightarrow$

$$k = \frac{\epsilon p_{max}}{n}$$

Alternatively, can use the 1/2 approximation algo and get $p_A \leq p^*$

Pseudo-polynomial algorithms and FPTAS

Definitions: $\max(I)$: the maximum number in the input I . $C^*(I)$: The optimal solution for input I

THM: If a problem Π over \mathbb{N} has an FPTAS $A(\epsilon)$, and if for each input I : $C^*(I) \leq p(\max(I))$ for some polynom p then the problem Π has a pseudo-poly algorithm.

23/1/06

PTAS for MIS on planar graphs

Generalization of the Small-Separator THM: For a planar graph G with n vertices and natural $1 \leq x \leq n$, G has a separator S of size $|S| = O(\frac{n}{\sqrt{x}})$ s.t. removing S cuts the graph into disjoint and siconnected $\{A_i\}$'s of size $|A_i| \leq x$. S can be found with complexity $O(n \log n)$

PTAS algorithm for MIS:

1. Use the generalization, and find $\{A_1, \dots, A_m\}$ disjoint with no edges between them, s.t. $|A_i| \leq x$
2. For each A_i , find an MIS $M_i \leftarrow MIS(A_i)$
3. Return $M = \bigcup M_i$

UDG - Unit Disk Graph

A graph is a UDG if it can be represented as points in the plane s.t. two nodes are connected iff their distance $\leq \sqrt{2}$.

PTAS for MIS on UDG in a Geometric Representation

Note that if all points are in a $k \times k$ square, the problem is poly.

Also note that if all points are in a strip of width k , the problem is also poly (by dynamic programming)

Algo.:

1. Cut the graph into strips of width 2
2. Repeat $k+1$ times:
3. In attempt l :
 - Remove any strip $i = l \bmod (k + 1)$
 - We now have “wide” strips of width $2k$
 - For each such strip, find an exact MIS (dynamic programming)
 - Define $M_l \leftarrow \bigcup$ MIS found in all strips
4. Return the biggest M_l found

PTAS for an MIS on UDG Without the Geometrical Representation

Actually, this algorithm only assumes $G \in \mathcal{G}$, when \mathcal{G} is the family of graphs s.t.:

Given G from the family, and $\epsilon > 0$, there exists a poly procedure Π_ϵ which finds a set of vertices W in G s.t.:

1. $MIS(W)$ can be computed in poly time
2. $|MIS(\Gamma(W))| \leq (1 + \epsilon)|MIS(W)|$
3. $G - \Gamma(W) \in \mathcal{G}$

Algorithm $A_\epsilon(G)$:

1. Let $\epsilon' \leftarrow \frac{\epsilon}{1-\epsilon}$
2. Call $\Pi_{\epsilon'}$ and get its W
3. $M_1 \leftarrow MIS(W)$
4. For $G' = G - \Gamma(W)$, recursively compute $M_2 \leftarrow A_\epsilon(G')$
5. Return $M_A = M_1 \cup M_2$

30/1/06

Greedy Algorithm for Hitting Set

Input: $U = \{u_1, \dots, u_m\}, S = \{s_1, \dots, s_n\}, S_i \subseteq U$

Output: Smallest set $X \subseteq U$ s.t. $\forall i, X \cap s_i \neq \emptyset$

(A similar problem: Set-Cover: Find a smallest set $\mathcal{S} \subseteq S$ s.t. $\bigcup_{s \in \mathcal{S}} s = U$)

Greedy Algorithm:

1. $X \leftarrow \emptyset$
2. While $S \neq \emptyset$, do:

- Choose an element $u \in U$ which hits the maximal number of sets in S
- $X \leftarrow X \cup \{x\}$
- Remove from S all sets which include u

This algorithm gives a logarithmic approximation

There's also a weighted version, in which every element has a cost c_i , and we're looking for a set with minimal cost (instead of a smallest set). Greedy algorithm: Define $\rho_i = \frac{\text{num hits by } u_i}{c_i}$, and choose by maximal ρ_i