Approximation Algorithms

NP-complete problems likely to be hard to solve.
NP-complete problems usually have an associated optimization problem.
As these problems are NP-hard, it is unlikely that we can get a fast algorithm to solve them.

Approximation Algorithms

Example: Travelling Salesman Problem.
Optimization Problem related to TSP:
Given a graph (V, E), where there are weights associated with each edge.
Problem: Find a tour (cycle) going through all the vertices (and returning to the starting vertex) with minimal weight, that is, the sum of the weights of the edges in the tour should be minimal among all such possible tours.

This problem is NP-hard. So we are unlikely to get a fast (polynomial time) algorithm to solve it.

Approximation Algorithms

Suppose the optimal tour costs $5000, but it takes 1 year for your travel agent to find the optimal tour.

On the other hand, suppose your travel agent can find a tour costing $5100, within 5 minutes.

For many purposes, this may be good enough.
[Also note that we may have costs associated with travel agent's time!]

So often, an approximate answer to an optimization problem may be good enough.
In this and next lecture we will look at some of the algorithms for "approximately" solving some NP-hard optimization problems.

Minimum Vertex Cover:

Recall the Vertex Cover Problem.

Given a graph $G = (V, E)$. V' is a vertex cover for G, iff $(\forall (u, v) \in E)[u \in V' \text{ or } v \in V']$.

The NP-complete problem related to the vertex cover question was:

Question: Does there exist a vertex cover of size at most K?

The optimization question is: Find vertex cover V' with minimum size.

The optimization problem is NP-hard. Thus it is not likely that one can find a minimum vertex cover for a graph in polynomial time.

However, we can do a reasonable approximation.

Vertex Cover

Input: $G = (V, E)$

Output: A vertex cover V' 'which is not too large'

Let $V' = \emptyset$.

Let $M' = \emptyset$. (M' is used only to give a proof that V' is not too large).

For each $(u, v) \in E$, do

If ($u \notin V'$ and $v \notin V'$) Then,

let $V' = V' \cup \{u, v\}$;

let $M' = M' \cup \{(u, v)\}$;

Endif

Endfor

End Vertex Cover

Clearly, the algorithm runs in polynomial time.
Suppose the size of the minimum vertex cover is m.
Then we claim that size of V', as given by the above algorithm is at most $2m$.
Hence the output of the above algorithm is within a multiplicative factor of 2 of optimal.
Proof of claim: Note that M' forms a matching.
Thus any vertex cover of G must have at least size(M') number of vertices.
Since number of vertices in V' is 2$*$size(M'), the size of vertex cover found by the algorithm is at most two times the optimal.

Approximation Algorithms in General

The vertex cover example illustrates that even though an optimal solution may be difficult to find, a reasonable approximation to optimal may be easy.

There are in general two kinds of approximations one can look for:

(1) Approximations which are within a constant 'additive' factor of optimal.

That is AlgoAnswer $\leq$ optimal + constant

(2) Approximations which are within a constant 'multiplicative' factor of optimal.

That is AlgoAnswer $\leq$ constant * optimal

For multiplicative factor approximations we also consider whether one can get arbitrarily close to 1.

Planar Graph coloring

Graph coloring: Color each vertex of the graph so that no adjacent vertices have the same color.
Planar Graphs: Graphs which can be drawn on a plane without any edges crossing each other. (Here we only consider simple graphs, which do not have parallel edges or self-loops)

It can be shown that every planar graph is 4 colorable.
Deciding whether a planar graph is 3 colorable or not is NP complete.
Thus finding an optimal (minimum) number of colors by which a planar graph can be colored (without adjacent vertices having same color) is NP-hard.
However we have an easy approximation algorithm for it.
Every planar graph is 4 colorable! So we are within an additive constant factor of optimal.

In a simple connected planar graph (graph which does not have parallel edges or self loops):
Suppose e is the number of edges, f is the number of faces, and v is the number of vertices.
(a) $e - f = v - 2$.
This can be proved by induction on the number of edges.
For a tree, the above holds as the number of vertices in a tree is one more than the number of edges, and the number of faces in a tree is 1.
For a planar graph with a cycle, if we delete one edge from a cycle, then the number of faces is reduced by one. Thus, by induction, the formula holds.
(b) $f \leq 2e/3$.
As each face is surrounded by at least three edges, and each edge is shared between two faces, we immediately have the above result.

From (a) and (b) we immediately have:
$v - 2 \geq e/3$.
Thus, $e \leq 3v - 6$.
Thus, average degree of a vertex is < 6.
Hence, there is a vertex v of degree at most 5.
By induction color rest of the graph using six colors. Then color vertex v, using the color not used by its neighbours.
Hence, any planar graph can be colored using at most 6 colors.

Slight modification of the proof: By considering the neighbours of v more carefully, allows us to color any planar graph using at most 5 colors.

— 4 coloring proof is difficult.

Bin Packing

Bin Packing Problem is as follows:
Instance: A set A of n objects, $a_1, a_2, \ldots, a_n$ and corresponding sizes $\text{size}(a_1), \text{size}(a_2), \ldots,$. We are also given a bin size L.
Question: What is the minimum number of bins needed to fill all the objects.
Here, we require that the sum of sizes of objects assigned to a bin must not exceed the bin size.
The above problem is NP-hard (Exercise. Hint: reduce partition to bin-packing).
We consider some easy approximation algorithms for bin-packing.

First Fit: Index the bins $B_1, B_2, \ldots$. Objects are considered for placement in order $a_1, a_2, \ldots$.
Suppose just before placing object a_i, the bin B_j is f_j full (i.e. the sum of the sizes of objects already assigned to B_j is f_j).
To place a_i, find the least j such that B_j has enough empty space to fit a_i (that is $L - f_j \geq \text{size}(a_i)$). Place a_i in B_j.

Best Fit: Similar to First Fit, except that we use the bin B_j such $L - f_j$ is minimized (along with the condition that $L - f_j \geq \text{size}(a_i)$).

First Fit Decreasing: First sort the objects in non-increasing order of sizes. Then use first fit.

Best Fit Decreasing: First sort the objects in non-increasing order of sizes. Then use best fit.

Suppose I is an instance of bin packing problem.
Suppose $opt(I)$ denotes the optimal number of bins needed to pack objects in instance I.
Let $FF(I)$, $BF(I)$, $FFD(I)$, $BFD(I)$ denote the number of bins used by First Fit, Best Fit, First Fit Decreasing, and Best Fit Decreasing, respectively.

Theorem: $FF(I) \leq 2 * opt(I) + 1$. Similar result applies to the other three algorithms.

Proof: Note that at the end of the first fit algorithm, there cannot be two non-empty bins which are filled less than 50%.
Thus all, except may be one, bins are filled at least 50%.
Therefore the number of bins used by the above algorithm is

$$\leq \frac{2\Sigma_{i=1}^{n}\text{size}(a_i)}{L} + 1$$

Since the optimal algorithm has to use $\frac{\Sigma_{i=1}^{n}\text{size}(a_i)}{L}$ bins, we have our theorem. QED

Actually, much better bounds than the multiplicative factor of 2 can be shown. It can be shown that,
$FF(I) \leq 1.7opt(I) + 2$.
$BF(I) \leq 1.7opt(I) + 2$.
$FFD(I) \leq \frac{11}{9}opt(I) + 4$.
$BFD(I) \leq \frac{11}{9}opt(I) + 4$.
However, the proof is quite long and complicated. We will not do it in this course.

Knapsack

Instance: A set $A = \{a_1, \ldots, a_n\}$ of objects, with corresponding sizes, size(a_i), and values, value(a_i).
A knapsack of size L.
For a subset A' of A let, size(A') = $\Sigma_{a \in A'}$size(a) and value(A') = $\Sigma_{a \in A'}$value(a).
Problem: Find a subset A' of A, with size(A') $\leq L$, which maximses value(A').
Note that the knapsack problem is NP-hard (Hint: reduce partition to knapsack).

We will give an approximation algorithm for knapsack problem, which gives a solution with value at least $(1 - \epsilon)$ of optimal, for any fixed $\epsilon > 0$.

To this end, we first give a procedure, which given a set B of objects (with corresponding sizes and values) and a knapsack size K, outputs a subset B' of B.

This set B' will fit in the knapsack and has "large enough" value. (The goodness of B' is not with respect to multiplicative factor but something else. More on this later).

Inputs to the procedure are:

(1) A set B of m objects, (with corresponding sizes and values)

(2) A knapsack size K.

Output of the procedure: A subset B' of B, such that size$(B') \leq K$.

Procedure Select

1. Sort the objects in B by non-increasing order of value(b)/size(b)
   (i.e. by non-increasing order of value per unit size).
   Let this order be $b_1, b_2, \ldots, b_m$.
2. Let spaceleft $= K$.
   $B' = \emptyset$.
3. For $i = 1$ to m do
   If size(b_i) $\leq$spaceleft, then
   Let $B' = B' \cup \{b_i\}$.
   Let spaceleft=spaceleft$-$size(b_i).
   Endif
   End for
4. Output B'.

End

Lemma: Let a set of objects B (with corresponding sizes and values), and knapsack size K be as given in the algorithm for Select. Then Select outputs a subset B' of B such that
$(\forall B'' : \text{size}(B'') \leq K)[\text{value}(B'') - \text{value}(B') \leq \max\{\text{value}(b) : b \in B''\}]$
i.e. B' is worse off by value of at most "one element" of B''.
Proof: Let B'' be an arbitrary subset of B such that $\text{size}(B'') \leq K$.
Order the elements of B'' in order of non-increasing $\text{value}(b)/\text{size}(b)$:
$b^1, b^2, \ldots, b^l$.
If $B'' \subseteq B'$ then we are done.
Otherwise let j be the minimum number such $b^j \notin B'$.
This implies that at the time b^j was considered in the algorithm Select, we had $\text{spaceleft} < \text{size}(b^j)$. Thus,
$\text{value}(B') \geq$
$\Sigma_{i=1}^{j-1}\text{value}(b^i) + [K - \Sigma_{i=1}^{j-1}\text{size}(b^i) - \text{size}(b^j)] * [\text{value}(b^j)/\text{size}(b^j)]$
(since at the time, when b^j couldn't have been fit, all the elements in B' have value per unit size more than $\text{value}(b^j)/size(b^j)$).

Moreover,
$\text{value}(B'') \leq \Sigma_{i=1}^{j-1}\text{value}(b^i)+[K-\Sigma_{i=1}^{j-1}\text{size}(b^i)]*[\text{value}(b^j)/\text{size}(b^j)]$,
(since we had arranged the elements of B'' in non-increasing order of value per unit size).
Thus,
$\text{value}(B'')-\text{value}(B') \leq \text{size}(b^j)*[\text{value}(b^j)/\text{size}(b^j)] = \text{value}(b^j) \leq \max\{\text{value}(b) : b \in B''\}$. QED

Though, the algorithm Select is quite good, it may not still be a constant factor approximation algorithm (the single element value may be quite large).
So we consider the following modification, which gives us a subset A' of A such that $\text{value}(A')$ is nearly optimal.

Knapsack

Inputs:

(1) A set A of n objects (with corresponding sizes and values).

(2) A knapsack size L.

(3) An optimality factor r.

Output: A subset A' of A, such that size$(A') \leq K$, and value(A') is at least $(1 - \frac{1}{r})*$ optimal.

1. Let $S_1, S_2, S_3 \ldots$, be all subsets of A which have at most r members.

2. For each S_j, such that size$(S_j) \leq L$, let B_j be B' given by algorithm Select, when the inputs to Select are $B = A - S_j$ and $K = L - \text{size}(S_j)$.

   (Think of B_j as choosing the remaining elements, when we have already decided to chose S_j).

   Output $A' = S_j \cup B_j$, which maximizes value$(S_j \cup B_j)$.

End

Note that the algorithm runs in polynomial time (though it runs in time exponential in r, r is a CONSTANT).
Theorem: Suppose a set A of objects (with corresponding sizes and values), a knapsack size L and constant r are given.
Then the knapsack algorithm outputs a subset A', with size$(A') \leq L$, such that value$(A') \geq (1 - \frac{1}{r}) *$ optimal.
Proof: Consider any optimal subset A'' of A.
Let S_j be a subset of r elements of A'' with highest value (if A'' has size $< r$, then $S_j = A''$).
Thus we have (using the Lemma above),
value$(A') \geq$ value$(S_j \cup B_j) \geq$ value$(S_j) +$ [value$(A'' - S_j) -$ max $\{$value$(a) : a \in A'' - S_j\}$].
Since, for all $a \in A'' - S_j$, for all $a' \in S_j$, value$(a) \leq$ value(a'):
max $\{$value$(a) : a \in A'' - S_j\} \leq \frac{\text{value}(S_j)}{r} \leq \frac{\text{value}(A'')}{r}$.
Thus,
value$(A') \geq$ value$(A'') - \frac{\text{value}(A'')}{r} \geq (1 - \frac{1}{r}) *$ value(A''). QED

Traveling Salesman

Can every NP-hard optimization problem be approximated within a constant multiplicative factor?

The answer is no (assuming $P \neq NP$). For this we consider the Traveling Salesman problem.

Suppose by way of contradiction that $P \neq NP$, and we have an algorithm to approximate the traveling salesman problem within a factor of k (where k is a constant).

That is, given a weighted graph $G = (V, E)$, (with weight of an edge e given by $wt(e)$), the algorithm can, in polynomial time, give a tour (simple circuit going through all the vertices of G) C of all the vertices in G, such that $wt(C) \leq k * opt$, where opt is the weight of the optimal tour.

We then show how to solve Hamiltonian circuit problem in polynomial time. (This will lead to a contradiction).

Suppose an unweighted graph $G = (V, E)$ is given.
Form a weighted graph G' as follows.
G' has all the edges, with $wt(e) = 1$, if $e \in E$, and $wt(e) = (k+1)n$, if $e \notin E$.
Note that,
G has a Hamiltonian circuit iff
G' has a tour of weight $\leq n$ iff
G' has a tour of weight $\leq k * n$.
Thus, G has Hamiltonian circuit iff the approximation algorithm for Traveling Salesman problem gives (on input G') a tour of weight at most $k * n$.
Since the above reduction can be done in polynomial time, along with the polynomial time approximation algorithm for Traveling Salesman problem, we have a polynomial time algorithm for Hamiltonian Circuit. A contradiction. QED

Traveling Salesman Problem with Triangular inequality

Suppose the weights of the edges of a graph G satisfy:
$wt(u, v) + wt(v, w) \geq wt(u, w)$ (triangular inequality).
Then one can, in polynomial time, find a tour C of G such that $wt(C) \leq 2 * wt(\text{optimal tour})$.

The idea is as follows. Given a graph G, form the minimum weight spanning tree of G. Note that this can be done in polynomial time.
Note that no tour of G has weight less than the weight of minimum weight spanning tree.
Duplicate each edge in this spanning tree, forming a (non-simple) graph G', with each vertex having even degree.
Thus, graph G' has an Euler circuit (which can be found in polynomial time).
This circuit goes through all the vertices of G and has weight at most twice the weight of the minimum spanning tree.

Now we can get a tour of weight at most the weight of the above circuit, using the following lemma.

Lemma : Suppose a (not necessarily simple circuit) C is given which goes through all vertices of G. Then one can construct a tour going through all the vertices of G, which has weight no more than the weight of C.
Proof: Use the following trick:
Suppose $C = C_1 u e_1 v e_2 w C_2$, where v is a repeated vertex.
Form another circuit C' as follows:
$C' = C_1 u e_3 w C_2$, where e_3 is the edge between u and w. Note that weight of C' is no more than the weight of C (due to triangle inequality) and C' goes through all the vertices of G.
Repeating the process for all repeated vertices, gives us the lemma.
QED