Coping with NP-completeness

So far...

* Techniques for writing (hopefully) efficient algorithms (greedy, dynamic programming, network flows, linear programming).

* Techniques for showing that problems cannot be solved efficiently (if P != NP, then every NP-complete problem D does not belong to P).

* Traditional point of view:
  - P = "easy"
  - NP-complete = "hard"

* But:
  - definition of NP-completeness based on worst-case analysis (maybe inputs encountered in practice are rarely worst-case)
  - for "real-world" input sizes (>= 10^9), even n^2 runtime is inefficient!

Approximation algorithms

Not possible to solve NP-hard problems *exactly* and *in polytime* (unless P = NP).

In practice, sometimes sacrifice on efficiency: use exponential-time algorithm and hope inputs don't trigger worst-case behaviour. Particularly useful on restricted families of inputs:

* A problem that's hard in general may be easy for inputs of a certain type. For example:
  - 2SAT.
  - Independent set on a tree.
  - Many graph problems are easy on trees or other restricted kinds of graphs. Others not (e.g., planar 3-colouring still NP-complete).

Sometimes sacrifice on exactness, particularly for optimization problems: instead of searching for *best* solution, settle for "good enough" solution. But what does that mean exactly?

For minimization problems, let OPT(x) be the minimum _value_ of any solution for input x. Suppose we have an approximation algorithm that generates solutions with approximate value A(x). By definition, OPT(x) <= A(x) for all inputs (since OPT(x) is minimum).

The "approximation ratio" of our algorithm is a function r(n) such that
    A(x) <= r(n) * OPT(x)   for all n and all inputs x of size n,
i.e., approximation ratio gives a bound on how much larger than optimum our approximate value might be -- it gives a guarantee that approximate values cannot be "too" large compared to optimum.

Vertex Cover:

* Approx algo 1: greedy strategy -- next week's tutorial...

* Approx algo 2:
  Repeatedly pick an edge and put both endpoints in C, then remove all edges incident on the two endpoints, until no edge remains. Then,
  |C| <= 2 * OPT
  because all covers include at least one endpoint from every edge in C (all edges in C are disjoint, i.e., with no endpoint in common) so in particular, OPT >= |C|/2. This shows approximation ratio <= 2.
  To show approximation ratio = 2, need an example where algorithm performs that badly -- in this case, use n disjoint edges! Algorithm returns 2n endpoints but n of them are sufficient.

Q: In general, how can we compute ratio without knowing OPT?
(Particularly for NP-hard problems, like VC, TSP, bin packing?)
A: Use a lower bound. Find another value LB that's easy to compute and for which you can prove LB <= OPT and A <= r * LB. For example...

* Approx algo 3:
  - Create linear program from input graph:
    variables: x_1,...,x_n (one for each v_i in V)
    obj. function: minimize x_1 + ... + x_n
    constraints: 0 <= x_i <= 1 for all i
    x_i + x_j >= 1 for each (v_i, v_j) in E
    (As an Integer Program -- domain for each x_i = Z -- this is completely equivalent to MinVertexCover, including NP-hardness.)
  - Compute optimal solution to linear program: x*_1, x*_2, ..., x*_n (linear program "relaxation": allow real values for variables).
  - Create cover as follows:
    for each v_i in V, put v_i in C iff x*_i >= 1/2.
    (C is a cover because constraint x_i + x_j >= 1 guarantees at least one of x*_i, x*_j >= 1/2 for each edge (v_i,v_j).)

  Approximation ratio?
  Consider minimum vertex cover C'. For i = 1,...,n, let x'_i = 1 if v_i in C'; x'_i = 0 otherwise. x'_1, ..., x'_n is a solution to linear program that satisfies all constraints with 0-1 values so
  |C'| = \sum x'_i >= \sum x*_i
  where x*_i is optimal solution to linear program with no restriction on values, so guaranteed to be at least as small as any other solution, including those with additional restrictions.
  For i = 1,...,n, let x~_i = 1 if x*_i >= 1/2; x~_i = 0 othewise.
  Then, for each i, x~_i <= 2 x*_i so
  |C| = \sum x~_i <= 2 \sum x*_i <= 2 |C'| (by equation above)
  Hence, |C| is no more than twice the size of a minimum vertex cover.

How well can problems be approximated?
Even though all NP-complete problems "equivalent" to each other (in one sense), approximation ratios for corresponding optimization problems all over the place.

* VC: constant approx ratio 2.
* Set Cover (in textbook): approx ratio log n (not constant but limited).
* Knapsack: approx ratio 1+epsilon in time O(n^3/epsilon), for all constants epsilon in (0,1]!
* TSP: no finite ratio, unless P=NP!

Traveling Salesman Problem (TSP):

* Given graph G with edge weights w(e), find a "tour" (Hamiltonian cycle) with minimum total weight.

* NP-hard: no polytime solution.

* NP-hard to approximate with constant ratio:
  Suppose we have an algorithm with approx ratio C(n), i.e., guaranteed to find tour with total weight <= C * OPT. We show how this algorithm could be used to solve an NP-hard problem.
  Given an input G to HamCycle problem, construct an instance of TSP G' as follows: on the same vertex set as G, put in all possible edges with w(e) = 1 if e in G, w(e) = Cn+1 if e not in G.
  Any tour in G' that uses only edges with weight 1 has total weight n; any tour in G' that uses at least one edge with weight Cn+1 has total weight > Cn.
  If G contains a Ham cycle, then G' contains a tour with total weight n; if G does not contain a Ham cycle, then all tours in G' have total weight > Cn+1 (they must contain at least one edge not in G).
  So now, run approx. algorithm on G'. If algo returns answer with total weight <= C*n, then G contains a Ham cycle; if algo returns answer with total weight > Cn+1, then G does not contain a Ham cycle. As this solves the Ham cycle problem, approx algorithm cannot run in polytime.

For Next Week
* Readings: Sections 9.2.3, 9.2.4, 9.2.5.
* Self-Test: None for this week...