CS 466/666 Notes
More Coping with NP- Hardness and Completeness

We will briefly discuss three approaches to handling NP-hard optimization problems, in addition to approximations with guaranteed bounds. There are

i) randomization (see section 35.4)
ii) derandomization
    and
iii) parameterization (often we face NP-hard problems with two parameters, it may be possible to solve them in time linear in one, though exponential in the other).

Randomization
First consider randomization and the idea of a randomized $\rho(n)$ approximation algorithm that has an expected approximation ration of $\rho(n)$ (or better) for all inputs. A key point here is that the method should be expected to work well, on any input. We illustrate the idea by considering a very simple example: Max-3-CNF satisfiability. That is to satisfy the maximum number of clauses in a 3-CNF expression (we take from that that each clause contains three distinct literals … x and $\neg x$ are distinct, but of course having them both in a clause simple makes it true).
A natural approach would be to choose the literal that occurs most often and set it to true. Another idea would be to find maximum difference in number of occurrences of and over all values of i; then make a "wise" assignment. These ideas tend to work well, but bad cases can be constructed. A reasonable baseline for further improvements can be constructed by making a random assignment to each variable, this leads to an 8/7 approximation. The result also tells us 7/8 of the clauses in any 3-CNF expression can be satisfied.
Proof:
Let Y_i= 1 if {clause i is satisfied} else 0
There are 3 literals in clause i, so a random assignment sets the clause to true, with probability at least 7/8 (if a variable and its negation both occur in the same clause, the clause must be true). So if Y is the number of the m clauses satisfied:

$$E[Y] = E[\Sigma Y_i] = \Sigma Y_i = 7m/8$$

Given this baseline one can try modifications in the hope of making improvements. Remember that on any given input, what you care about is getting a good answer in the time you have available.

Derandomization
Sometimes we have a good technique that involves randomization and we would like to remove the probabilistic aspects of its behavior. An example of this might be moving from standard binary search trees which work well on a random ordering of insertions to some sort of balanced trees such as AVL trees that have a guarantee of good behaviour. The 3-Sat example above gives us to try a simple scheme.
Choose and arbitrary variable, call it a. We will assign a to be true or false, whichever is most effective for our purpose. We might say "whichever makes the most clauses true", that is fine when we start but needs a bit of refinement. Note that making an assignment of a makes some clauses true while reducing the number of "open variables" in other clauses, at some stage as the process continues such an assignment will make some clauses false, while hopefully making more of them true. So we go back to thoughts of the probabilistic approach. A random assignment makes a clause with 3 literals true with probability 7/8. If we make a random assignment of a variable, with either

that variable or its negation as literals in a clause, the clause is either made true or reduced to a clause with 2 literals … with equal probability. So the clause becomes true with probability 1 or with probability 3/4 (the latter case, if our literal is false). Continuing the idea and moving from probabilities to “weights”, we say:

- A clause already made true by assignments has weight 1.
- A clause with 3 literals has weight 7/8, making a literal true increases this by 1/8, making it false decreases by 1/8.
- A clause with 2 literals has weight ¾, making a literal true increases this by 1/4, making it false decreases by1/4.
- A clause with 1 literal has weight 1/2, making a literal true increases this by 1/2, making it false decreases by 1/2.
- A clause with no literals left (i.e. all have been made false) has weight 0, and there is nothing we can do.

This means that if making an assignment of a increases (or decreases) the sum of all clause weights by δ, making the reverse assignment of a would change the weight sum by the same amount in the opposite direction. So we make a either true or false, whichever makes the sum of all weights greater, this means the weight sum never decreases when a variable is assigned a value. So when all variables have been assigned, at least 7/8 of them are true.
Observe that having first linked clauses containing a common literal in a doubly linked list, the entire process can be done in linear time.
Curiously, although this means we can satisfy 7/8 of the clauses in any 3-Sat problem, Håstad has shown that getting a better ratio relative to the optimal assignment of any expression, is NP-Hard.

Parameterized Complexity
Many NP-complete decision problems, and NP-hard search problems, are expressed in terms of more than one parameter. For example, we could be asked whether an n node graph has a clique of size k. Although the problem is NP-complete if k is arbitrary, it is obvious the problem can be solved in $O(n^k)$ time by brute force. There are also cases in which $O(n^c 2^k)$ or even $O(n^c + 2^k)$ solutions exist. This parameterized complexity point of view can be very useful and lead to a guarantee of an exact solution in time polynomial (perhaps even linear) in the size of the input plus an exponential term in a much smaller parameter. Formally we say a problem is **fixed parameter tractable** if there exists a solution that runs in time $O(f(k)\, n^c)$ where n is the input length, k is a parameter supplied in the problem and c is a constant.
It quickly follows that if an optimization problem in NP has a fully polynomial time approximation scheme, then it is a fixed parameter tractable. (Simply get a (1+1/k) approximation, this guarantees the error so small that it must be the optimal solution)
We will look at only one example, the vertex cover problem, but see an approach that has been applied in many cases. So, we are given an n node, e edge graph and asked for a k-vertex cover.
The first phase (called **kernelization**) is simply a matter of scanning the entire graph and reducing the problem to one of size that depends only on k.
So, for the problem at hand, we observe that for any given node, either it or all its neighbours must be in a vertex cover. Hence, every node of degree greater than k must be in the k-cover, as omitting it alone would make the cover too large.
Having removed all such nodes, we are left looking for a k’ cover. (Assume we had k-k’ nodes of degree greater than k.) Next observe than if there is a k cover for the original graph, we are to choose at most k’ more, and none of these nodes is of degree greater than k. Hence the remaining graph has

at most kk’ edges, and so at most 2kk’ nodes of degree more than 0 (though the latter bound is easily improved). In any case we have drastically reduced the size of the problem.
From the argument above we can try all possible subsets of k’ of these nodes and solve our problem. This would give an O(n + e+ $(2kk')^{1+k'}$) or so method, but reducing the k term even farther can dramatically extend the range in which the method is useful.
The idea is to note that, for any remaining edge, at least one of the nodes incident with that edge must be in the k’ cover. We will build a complete binary tree of height k’, in which each path corresponds to a k’ cover. We start by choosing an arbitrary edge of the reduced graph, each of its children is labelled with one end of this edge. The process continues at each child, where an arbitrary edge, neither of whose ends appears on the path from the root. Again each of the children of this tree node is labeled with one end of the selected edge. If any path in this tree, of height k’, covers all edges, we have a solution, otherwise there is no solution. Clearly we take at most O(kk’$2^{k'}$) time, but with care we can get this portion to run in time O($2^{k'}$). This leads to the desired O(n+e+2^{k}) method.
With a bit more work the method can be improved even more, to about O(n+e+$1.28..^{k}$), which effectively triples the value of k that can be accommodated in the same length of time. There exist implementations that can solve the problem for all graphs with a few billion nodes and k up to 200 or so.