**Introduction to discrete optimization.**

The difficulty of sifting through large amounts of data in order to make an informed choice is ubiquitous in today’s society. One of the promises of the information technology era is that many decisions can now be made rapidly by computers, from deciding inventory levels, to routing vehicles, to organizing data for efficient retrieval. The study of how to make decisions of these sorts in order to achive some best possible goal, or objective, has created the field of *discrete optimization*.

Unfortunately, most interesting discrete optimization problems are NP hard. Thus, unless P = NP, there are no efficient algorithms to find optimal solutions to such problems, where we follow the convention that an efficient algorithm is one that runs in time bounded by a polynomial in its input size.

**NP. NP-hard.NP-complete.**

NP stands for Non-deterministic Polynomial time.

This means that the problem can be solved in Polynomial time using a Non-deterministic Turing machine (like a regular Turing machine but also including a non-deterministic "choice" function). Basically, a solution has to be testable in poly time. If that's the case, and a known NP problem can be solved using the given problem with modified input (an NP problem can be reduced to the given problem) then the problem is NP complete.

The main thing to take away from an NP-complete problem is that it cannot be solved in polynomial time in any known way. NP-Hard/NP-Complete is a way of showing that certain classes of problems are not solvable in realistic time.

An answer to the P = NP question would determine whether problems that can be verified in polynomial time, can also be solved in polynomial time. If it turned out that P ≠ NP, it would mean that there are problems in NP (such as [NP-complete](http://en.wikipedia.org/wiki/NP-complete) problems) that are harder to compute than to verify: they could not be solved in polynomial time, but the answer could be verified in polynomial time.

**Approximation algorithms. Approximation ratio.**

If P != NP, we can’t simultaneously have algorithms that (1) find optimal solutions (2) in polynomial time (3) for any instance. The most common approach is to relax the requirement of finding an optimal solution, and instead settle for a solution that is “good enough”, especially if it can be found in seconds or less. Various types of heuristics (technique designed for [solving a problem](http://en.wikipedia.org/wiki/Problem_solving) more quickly when classic methods are too slow) have been studied, leading to the conclusion that these techniques often yield good results in practice.

An α-approximation algorithm for an optimization problem is a polynomial-time algorithm that for all instances of the problem produces a solution whose value is within a factor of a α of the value of an optimal solution.

For an α-approximation algorithm, we will call α the performance guarantee of the algorithm. In the literature, it is also often called the approximation ratio or approximation factor of the algorithm.

The **approximation ratio** (or **approximation factor**) of an algorithm is the ratio between the result obtained by the algorithm and the optimal cost or profit. When the approximation ratio is close to 1, it is often more useful to look at the **approximation error**, which is defined as the approximation ratio minus 1. So an algorithm that always got within 1.01 of the optimal cost or profit would have a 1% approximation error.

**Vertex Cover Problem.**

The minimum vertex cover problem on a graph asks for as small a set of vertices as possible that between them contain at least one endpoint of every edge in the graph. It is known that vertex cover is NP-hard, so we can't really hope to find a polynomial-time algorithm for solving the problem exactly. Instead, here is a simple 2-approximation algorithm   
(the pseudo-code I used)

APPROX VERTEX COVER(G)  
C=∅  
E'= G.E  
while E'≠ ∅

let (u,v) be an arbitrary edge of E'

C = C ∪ {u,v}

remove from E' every edge incident on either u or v

return C

To show that this gives a 2-approximation, consider the set E' of all edges the algorithm chooses. None of these edges share a vertex, so any vertex cover must include at least |E'| vertices. The algorithm marks 2|E'| vertices.

Or:

One can find a factor-2 [approximation](http://en.wikipedia.org/wiki/Approximation_algorithm) by repeatedly taking both endpoints of an edge into the vertex cover, then removing them from the graph. Put otherwise, we find a [maximal matching](http://en.wikipedia.org/wiki/Maximal_matching) M with a greedy algorithm and construct a vertex cover C that consists of all endpoints of the edges in M. In the following figure, a maximal matching M is marked with red, and the vertex cover C is marked with blue.

C:\Users\Alexa\Desktop\200px-Vertex-cover-from-maximal-matching.svg.png

The set C constructed this way is a vertex cover: suppose that an edge e is not covered by C; then M ∪ {e} is a matching and e ∉ M, which is a contradiction with the assumption that M is maximal. Furthermore, if e = {u, v} ∈ M, then any vertex cover – including an optimal vertex cover – must contain u or v (or both); otherwise the edge e is not covered. That is, an optimal cover contains at least one endpoint of each edge in M; in total, the set C is at most 2 times as large as the optimal vertex cover.