

## Dealing with NP-completeness

### Review

- Recall that a decision problem (i.e. language) is **NP**-complete if it is in **NP**, and every language in **NP** reduces to it in polynomial time.
- As a consequence, if any **NP**-complete language is decidable in (deterministic) polynomial time, then  $\mathbf{P} = \mathbf{NP}$ ; and if any **NP**-complete language is not decidable in (deterministic) polynomial time, then  $\mathbf{P} \neq \mathbf{NP}$ .
- It is widely believed that  $\mathbf{P} \neq \mathbf{NP}$ , and a consequence of this belief is that **NP**-complete problems are not believed to be decidable deterministically in polynomial time.
- Note that although polynomial time includes running times such as  $n^{100}$ , which are in practice so large as to be intractable, the assumption that  $\mathbf{P} \neq \mathbf{NP}$  implies that, whatever particular definition of “efficient” one is working with (e.g. linear time, or quadratic time), there is certainly no efficient algorithm for any **NP**-complete problem.
- As a great number of natural problems are **NP**-complete, this leads us to the question: what can be done to “solve” a problem “efficiently” once it has been proved **NP**-complete. Clearly we are going to have to change the definition of “solve”, or change the definition of “efficient”, or change the definition of the problem.
- Below is an overview of some techniques that have been successful in this regard.

### Approximation Algorithms

- Many **NP**-complete problems have a natural corresponding optimization problem. For example, one is generally not interested in the question “does  $G$  have a vertex cover of size  $k$ ?” so much as “find a minimum vertex cover of  $G$ ”. Other problems we have discussed that have natural, underlying optimization problems include SET-COVER, CLIQUE, INDEPENDENT-SET, KNAPSACK, and TSP.
- In this case, as we have seen, we can try to find a solution that is (provably) within a particular bound of the optimal solution. The degree to which we can achieve this goal (i.e. how close we can get) depends on the particular problem: although **NP**-complete problems are equivalent with respect to polynomial-time solvability, they are not all equivalent with respect to approximability.
- Some of the results we have seen:

- VERTEX-COVER: We've seen an extremely simple 2-approximation algorithm (find a maximal matching and return all endpoints of the edges in the matching). Notice that not only is this algorithm polynomial-time, it is very fast, running in nearly linear time. It is interesting that the approximation ratio of 2 is essentially the best known for VERTEX-COVER: there is no known algorithm achieving a constant approximation better than 2. A lower bound of about 1.36 has been proved on the approximability of VERTEX-COVER, assuming  $\mathbf{P} \neq \mathbf{NP}$  (i.e. if  $\mathbf{P} \neq \mathbf{NP}$  then there is no polynomial-time algorithm achieving a ratio better than 1.36). The question of the true approximability of VERTEX-COVER (somewhere between 1.36 and 2) remains an interesting open problem.
- SET-COVER: We've seen a simply greedy algorithm that achieves a ratio of  $H(n) \sim \ln n$ , where  $n$  is the number of elements in the universe. There is a corresponding lower bound, up to a constant factor: no polynomial time algorithm achieves an approximation ratio in  $o(\ln n)$ . The question of the approximability of SET-COVER is therefore almost entirely resolved. Note that the greedy algorithm is also very fast, and useful in practice.
- KNAPSACK: We've seen that for any  $\varepsilon > 0$  we can achieve a  $(1 - \varepsilon)$ -approximation in time  $O(n^3/\varepsilon)$ , using dynamic programming and scaling. This is called an FPTAS. Again, even for quite small values of  $\varepsilon$  this algorithm runs in a reasonable amount of time, and is therefore applicable in practice as well as in theory. In some sense this is the best we can hope for, unless  $\mathbf{P} = \mathbf{NP}$ . Notice that an FPTAS is impossible for VERTEX-COVER or SET-COVER, as a consequence of the inapproximability results.
- TSP: in this case we saw that for every constant  $\alpha \geq 1$  (and in fact for any polynomial-time computable function  $\alpha : \mathbb{N} \mapsto \mathbb{N}$ ), there is no  $\alpha$ -approximation (or  $\alpha(n)$ -approximation) for TSP.

## Additional Assumptions

- In the case of inapproximable problems like TSP, or even if the achievable approximation ratio is not satisfactory, it may be possible to make some additional assumptions about the input, which corresponds to modifying the problem.
- In the case of TSP, we saw a 3/2-approximation algorithm for the related problem Metric TSP, where the distance function satisfies symmetry ( $d(i, j) = d(j, i)$ ) and the triangle inequality ( $d(i, j) \leq d(i, k) + d(k, j)$ ).
- Another restriction that may in some cases be reasonable is Euclidean-TSP, where the cities are coordinates in  $\mathbb{R}^d$  and the distance between two

cities is the Euclidean distance between the points:

$$d((x_1, \dots, x_d), (y_1, \dots, y_d)) = \sqrt{\sum_{i=1}^d (x_i - y_i)^2}$$

In this case a PTAS is possible.

- Some other assumptions one might make:
  - VERTEX-COVER:
    - \* The graph has constant or bounded degree
    - \* The graph is a tree (in this case the problem can be solved in polynomial time by dynamic programming)
  - SET-COVER:
    - \* Each element  $u$  appears in at most  $f$  sets (in this case an  $f$ -approximation is possible)
  - HAM-PATH:
    - \* The graph is acyclic (solvable in polynomial time by dynamic programming)

### Fixed-Parameter Tractability

- Recall that a brute force approach for VERTEX-COVER runs in time roughly  $O(n^k)$ . The idea of fixed-parameter tractability is to achieve a running time consisting of two terms, where the first term is polynomial in both  $n$  and  $k$ , and the second term is non-polynomial in  $k$  but does not contain  $n$ .

- In the case of VERTEX-COVER, there is an algorithm running in time

$$O(kn + 1.325^k k^2)$$

- The difference here is largely a matter of pragmatics. Consider an input with  $n = 100$  vertices and  $k = 50$ . The running time of the FPT algorithm is on the order of  $10^9$ , and if we assume  $10^9$  operations per second this requires about one second. On the other hand, the BF (for Brute Force) algorithm requires about  $10^{100}$  operations, corresponding to about  $10^{83}$  years. To get a feel for how large this is, compare it to the estimated age of the universe.

FPT :	1 second
BF :	$10^{83}$ years
Estimated age of universe:	$10^{10}$ years

- Thus even if we had started the Brute-Force algorithm at the moment of the Big Bang, it would not even be close to completing.

## Heuristics

- When all else fails, one can simply decide on an algorithm and “try it out”, i.e. see how it performs on a set of test cases. “Better” algorithms can be found by comparison of these empirical results, for example one may observe that a particular algorithm achieves a 5% smaller vertex cover, on average, on the test cases than another algorithm; or that it finds an optimal vertex cover and runs 5% faster on the test cases.
- Bear in mind that this should really be a last resort: the performance of an algorithm on a particular (and necessarily small) set of inputs may have no resemblance to its worst-case performance, unless there is reason to believe that the test cases really are indicative of the actual inputs that will be encountered.
- One way to give some sort of theoretical foundation to these types of algorithms is to consider the “average case”: if you pick an input of size  $n$  randomly and uniformly (i.e. all inputs of length  $n$  have equal probability) what is the expected running time (or approximation ratio, etc)? Alternately, instead of a uniform distribution one can consider each input  $I_j$  as having probability  $p_j$ , and bound the expected running time (or whatever) with these probabilities (the idea being to find a probability distribution that is “close” to what you expect to see in practice).