CSE5401 Lec. 3

Intractability (review)

1. The reduction method
2. A guide to hard problems
3. An example

Most of the problems we consider are believed to be intractable to solve optimally. In most cases, this belief is based on a conjecture that some <u>other</u> well studied problems are intractable. Depending on how broad or narrow the family of problems is, the conjecture may be stronger or weaker. For example, cryptography is usually based on the assumption that a specific "famous" problem is very hard on average — a strong assumption — whereas in the study of algorithms, we often get intractability based on the $P \neq NP$ assumption: that for <u>some</u> problem, there is a polynomial-time algorithm to verify solutions, but no polynomial-time algorithm to generate solutions on a worst-case input. Because the breadth of such polynomial-time verifiable problems is so great and the worst-case requirement is so much more demanding, this is a weaker assumption - more likely to be true.

Regardless of the assumption, practically speaking, we demonstrate intractability using <u>reductions</u>

Defn: We say that A polynomial-time reduces to B if there is an algorithm that uses a subroutine for B (at unit cost per invocation), that runs in polynomial time in the size of its input that solves A.

If we believe A is intractable and A polynomial-time reduces to B, then we conclude B is also intractable. Why? If we had a polynomial time algorithm for B, we could use it to implement the calls in the reduction, giving a polynomial-time algorithm for A. Then A would not be intractable.

Reductions are usually easier to design when the two problems are similar, and when the (presumed) intractable problem is relatively simple. So, it's helpful to have a selection of clean intractable problems at our disposal. For example, the following:

Clique: Given an undirected graph G, find a subgraph K that forms a "clique" ($\forall u,v \in V(K), [u,v] \in E(G)$) of maximum size (# vertices)

(A "packing" problem: $(u,v) \notin E(G)$ forbids using both)

Vertex Cover: Given an undirected graph G, find a set of vertices $S \subseteq V$ s.t. for every $(u,v) \in E(G)$, $u \in S$ or $v \in S$ and S is of minimum size

(A "covering" problem: must use u or v for every $(u,v) \in E$.)

3-coloring: Given an undirected graph G, find a coloring $\chi : V \to \{R, Y, B\}$ s.t. for every $(u,v) \in E(G)$, $\chi(u) \neq \chi(v)$.

(A "partitioning" problem: split G into three sets $\{v : \chi(v) = R\}, \{v : \chi(v) = Y\}, \{v : \chi(v) = B\}$ where (u,v) forbids packing u & v in the same component.)

Subset sum Given a set of integers $x_1, \ldots, x_n$ and a target integer t, find $S \subseteq \{1, \ldots, n\}$ s.t. $\sum_{i \in S} x_i = t$.

(An integer optimization problem.)

Hamiltonian path Given an undirected graph G, find a permutation $\pi : \{1, \ldots, |V|\} \to V$ s.t. for $i = 1, \ldots, |V|-1$, $(\pi(i), \pi(i+1)) \in E(G)$ — a path that visits every $v \in V$ exactly once.

(A "sequencing" problem, where we find a permutation.)

3D matching Given $H \subseteq U \times V \times W$ where $|U| = |V| = |W|$, find $M \subseteq H$ such that every $u \in U$, $v \in V$, $w \in W$ belong to exactly one $e \in M$.

(An "assignment" problem: every $u \in U$ must be assigned distinct $v \in V$ & $w \in W$ subject to $(u,v,w) \in H$.)

And finally, if all else fails

3SAT Given m clauses each consisting of three literals, which are either one of our n Boolean variables $\{x_1, \ldots, x_n\}$ or the negation of one of the variables, find an assignment of Boolean values to the variables such that in every clause, at least one literal is true.

These problems are NP-complete: any of them can be reduced to any of the others. They have a polynomial-time algorithm if and only if $P = NP$.

If we can reduce an NP-complete problem A to a problem B, B is NP-hard. But B might not be NP-complete: it is only NP-complete if we can verify solutions in polynomial time, e.g., that a presented set of vertices is a clique, vertex cover, etc. of a claimed size. But for the purposes of establishing intractability, NP-hardness is adequate. NP-completeness only means (a) that we can't be certain of the problem's intractability without establishing $P \neq NP$, and (b) that $P = NP$ is sufficient for there to exist a polynomial time algorithm. Some of the problems we'll consider will be NP-hard but not NP-complete. So, we'll focus primarily on the reductions that establish NP-hardness. Again, the reductions are generally easier the closer the problems are. For example, recall the Knapsack problem: given n items with weights and values, and a bound W on the total weight, find a set S of items with $\sum_{i \in S} w_i \leq W$ maximizing the total value, $\sum_{i \in S} v_i$.

We can reduce Subset Sum to Knapsack as follows: For each of the n values $x_1, \ldots, x_n$, we create an item with both weight and value equal to x_i. We set the bound W to the target value t. We use our subroutine for Knapsack to produce an optimal value subset, and if the value is $< t$, we return "infeasible" and otherwise we return the set of indices S.

Claim The algorithm solves Subset Sum, and runs in polynomial time (with unit-cost subroutine)

Proof: First, suppose there is a solution to the Subset Sum problem. Consider the set of indices S s.t. $\sum_{i \in S} x_i = t$. Since the items in the Knapsack instance have weight & value both equal x_i, the same set of indices S has weight $\sum_{i \in S} x_i = t$, and is therefore feasible, and attains value $\sum_{i \in S} x_i = t$. Therefore, the Knapsack subroutine must return a solution of value at least t. But, since we have that for all S', the weight equals the value, the value must be exactly t, so the returned set of indices satisfies $\sum_{i \in S'} x_i = t$. Conversely, consider any set S' returned by the Knapsack subroutine. We have already established that if it has value $< t$, the Subset Sum instance must not have been feasible. But otherwise, again, because the weight is equal to the value, which is $\leq t$, the set is indeed of value t, so again, $\sum_{i \in S'} x_i = t$. Thus in either case, we correctly solve Subset Sum. It is immediate that the algorithm runs in linear time. ▨

Most reductions require a more careful design of the instance(s) we pass to the subroutines, to replicate the constraints of the original problem