Self-reducibility

Last weeks we talked about some decision problems. As we previously mentioned, there are other problems that are more naturally "search problems": given input X, find a solution Y.

Examples:

- Given a propositional formula F, find satisfying assignment, if one exists.
- Given a graph G and a number k, find a vertex cover of size k, if one exists.
- Given graph G, find a Hamiltonian path in G, if one exists.
- Given graph G, integer k, find a clique of size k in G, if one exists.
- Given a set of numbers S and a target number t, find a subset of S whose sum equals t, if one exists.

Clearly, an efficient solution to a search problem would give efficient solution to the corresponding decision problem. So if the decision problem is NP-hard, it implies that the search problem is also "NP-hard" (in some generalized sense of NP-hard that contains search problems too) and does not have any efficient algorithm to solve it.

It's interesting to know that many search problems are only polynomially more difficult than corresponding decision problem, in the following sense: any efficient solution to the decision problem can be used to solve the search problem efficiently. This is called *self-reducibility.*

Example 1: CLIQUE-SEARCH
Given an Undirected graph G and a positive integer k, find a clique of size k in G, if one exists; special value NIL if there is no such clique in G.

Assumption: There is an algorithm $CL(G, k)$ that returns True iff G contains a clique of size k – *i.e.*, CL solves the decision problem.

WARNING! The argument for self-reducibility is NOT that "any algorithm that solves the decision problem must include a part that solves the search problem", as this is in fact not always true. For example, there is an algorithm that can determine whether or not an integer has any factors (*i.e.*, whether or not it belongs to COMPOSITES) without actually finding any of the factors (through some fairly involved number theory about properties of prime numbers).

In this case, for example, it would be a circular argument to assume that the algorithm CL must find a clique of size k in order to return whether or not such a clique exists. Instead, what we must do is show how to write a different algorithm that searches for a k-clique in G, by making use of the information provided by calls to CL.

Idea: For each vertex in turn, remove it iff resulting graph still contains a k-clique.
Details: Algorithm 1 solves the CLIQUE-SEARCH problem.

Algorithm 1: CLIQUE-SEARCH SOLVER

```
if not $CL(G, k)$ then
    return NIL // no $k$-clique in $G$
foreach vertex $v$ in $V$ do
    // remove $v$ and its incident edges
    $V' = V - \{v\}$
    $E' = E - \{(u, v) : u \in V\}$
    // check if there is still a $k$-clique
    if $CL(G' = (V', E'), k)$ then
        // $v$ not required for $k$-clique, leave it out
        $V = V'$
        $E = E'$
return $V$
```

Correctness: $CL(G = (V, E), k)$ remains true at every step so at the end, V contains every vertex in a k-clique of G. At the same time, every other vertex will be taken out because it is not required, so V will contain no other vertex. Hence, the value returned is a k-clique of G.

Runtime: Each vertex of G examined once, and one call to CL for each one, plus linear amount of additional work (removing edges). Total is $\mathcal{O}((n + 1) \times T(n, m) + n \times (n + m))$ where $T(n, m)$ is runtime of CL on graphs with n vertices and m edges; this is polytime if $T(n, m)$ is polytime.

Exercise: What happens if G contains more than one k-clique?

General technique to prove self-reducibility:

- Assume hypothetical algorithm to solve decision problem,

- Write algorithm to solve search problem by making calls to decision problem algorithm (possibly many calls on many different inputs),
- Make sure that search problem algorithm runs in polytime if decision problem algorithm does– argue at most polynomially many calls to subroutine are made and at most polytime spent outside those calls.

Example 2: HAMPATH-SEARCH
Given a graph G and vertices s,t, find a Hamiltonian path in G from s to t.

Hamiltonian path: a path that visits every vertex exactly once.

Idea 1: For each vertex in turn, remove it iff resulting graph still contains a Ham. path.

Problem: Every vertex must be in the path anyway, and this does not say where to put each vertex (which edges to use to travel through this vertex).

Idea 2: Remove s and its edges. Then consider each neighbour of s (must keep track of them separately), find one that has a Ham. path to t and remove it and its edges. Repeat until t is reached. Potential for exponential number of paths, but only polynomially many will be examined– with HP decision algorithm, only consider paths guaranteed to be Hamiltonian.

Idea 3: For each edge in turn, remove it iff resulting graph still contains a Ham. path – same as for CLIQUE above, except considering edges one-by-one instead of vertices.

Both ideas work.

Example 3: VERTEX-COVER-SEARCH
Given a graph G and an integer k, find a vertex cover of size k, if one exists (NIL otherwise).

Idea 1: Remove vertices one-by-one as long as the resulting graph still contains a vertex cover of size k.

Problem: If G contains a VC of size k, then $G - v$ (remove v and all incident edges) also contains a VC of size k, whether or not v is in the cover (unless $n = k$, trivial to solve)!

Idea 2: Check if $G - v$ contains a VC of size $(k - 1)$.

Algorithm 2: VERTEX-COVER-SEARCH SOLVER

```
if not $VC(G, k)$ then
    return NIL
$C = \{\}$ // the vertices in a vertex cover of $G$
foreach vertex $v \in V$, and while $k > 0$ do
    if $VC(G - v, k - 1)$ then
        $C = C \cup \{v\}$
        $G = G - v$
        $k = k - 1$
return $C$
```

Correctness: Loop invariant: G contains a VC of size k. At each iteration,

- if $G - v$ contains a VC of size $(k-1)$, then G contains a VC of size k that includes v: say C' is VC of size $(k-1)$ in $G - v$, then $C' \cup \{v\}$ is a VC of size k in G
- if G contains a VC C of size k that includes v, then $G - v$ contains a VC $C - \{v\}$ of size $(k-1)$; taking the contrapositive: if $G - v$ does not contain a VC of size $(k-1)$, then v does not belong to any VC of size k in G.

Runtime: $\mathcal{O}((n+1) \times T(n,m) + n \times (n+m))$– for each vertex v, we perform one call to VC in time $T(n,m)$ and compute $G - v$ in time $\mathcal{O}(n+m)$.

Optimization problems: Now let's focus on "optimization problems":

Some search problems with one or more numerical parameters naturally occur in practice in the form of optimization problems, *e.g.* MAX-CLIQUE, MIN-VERTEX-COVER, MAX-INDEPENDENT-SET, etc.

Optimization problems can also be polytime self-reducible by using decision problem algorithm to find optimal value of relevant parameter(s).

Example: MAX-CLIQUE
Idea: Given G, perform binary search in range $[1, n]$ by making calls to $CL(G, k)$ for various values of k, in order to find maximum value k such that G contains a k-clique but no $(k+1)$-clique. Then, use search algorithm to find k-clique.
This makes $\mathcal{O}(\log n)$ calls to CL.
(Note: Linear search for value of k would also be OK because search is in the range $[1..n]$ and input size $= n$.)

Similar idea would work for MIN-VERTEX-COVER, MAX-INDEPENDENT-SET, etc.

Back to decision problems:

Example: Show 3-coloring $\leqslant_p$ CNF-SAT.

(1)- Transformation: Given graph $G = (V, E)$ we create a formula F_G as follows:

- For each node $v_i \in V$, create one clause $(r_i \vee b_i \vee g_i)$
- For each edge $e = (v_i, v_j) \in E$, create three clauses: $(\sim r_i \vee \sim r_j)$, $(\sim b_i \vee \sim b_j)$, and $(\sim g_i \vee \sim g_j)$.

(2)- This transformation is polytime. Note that the size of F_G is comparable to the size of graph G.

(3)- G is 3-colorable if and only if F_G is satisfiable. (prove both $\Rightarrow$ and $\Leftarrow$)

Example: Show 3SAT $\leqslant_p$ Independent Set.

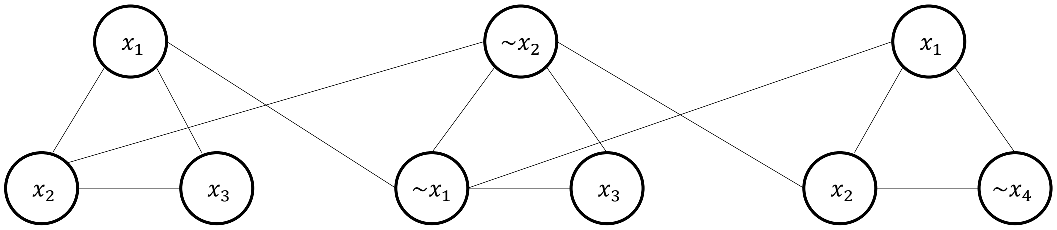
(1)- Transformation: Given a formula F, create a graph $G_F = (V, E)$ and a number k as follows:

- For each clause in F, G_F contains 3 vertices (one for each literal).
- Connect the 3 nodes corresponding to the 3 literals in a clause in a triangle.
- Connect each literal to all of its negations.

- Set $k =$ the number of clauses in F.

(2)- This transformation is polytime; the size of G_F is comparable to the size of F.

(3)- F is satisfiable if and only if G_F has an independent set of size k. (prove both $\Rightarrow$ and $\Leftarrow$)



$F = (x_1 \vee x_2 \vee x_3) \wedge (\sim x_1 \vee \sim x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \sim x_4)$