CSE202
6 – Dealing with NP-Complete Problems

TA: Joseph L.

I. Introduction

- As we saw in the previous section, some problems (NP-complete and harder) are *in general* not efficiently solvable, and are appropriately labeled as "intractable".
- In a real-world situation, one faced with a difficult problem may argue for the problem's difficulty by proving it NP-complete, justifying his/her team's inability to find an efficient solution.
- Sometimes, we would like to *attempt* to solve these problems regardless. This section discusses three approaches that one can take, and will feature example problems for the first two.
  - If we find our problem to be a *special case* of an intractable problem, then we may in fact have a polynomial algorithm. Some examples:
    - In boolean satisfiability, 2-SAT is efficiently solvable (whereas 3-SAT and above are not).
    - In graph-based problems, there may be efficient algorithms that work only for trees, or for graphs in which node degrees are constrained.
  - We may be able to approximate the solution (often with a greedy algorithm) with a mathematically bounded margin of error. For example, if we would like to maximize some quantity, we may be able to come up with an algorithm whose solution always yields at least half of the optimal quantity (and we'll have to prove that bound).
  - Sometimes, a problem may have solutions that would likely work well "in practice" despite an absence of mathematical guarantees. For example, suppose we would like to navigate a maze using some kind of search. If we know that the goal is usually near the starting point, then a breadth-first search would likely perform relatively well even though it is technically still an exhaustive search.

II. Specific Cases of Intractable Problems – Questions to Ask

- *What about this case makes the problem easier?* For example, if the input is a graph that's constrained to be a tree, then the lack of cycles is likely an important factor that enables options such as dynamic programming.
- *If the input size is constrained, how does this this simplify an otherwise-intractable problem?* As an extreme example, consider 1-SAT. In this problem, there is either no solution or exactly one, and each time we inspect a variable, we're able to rule out one value for it. We can't do this for 3-SAT due to the more complex interactions between variables (multiple clauses with multiple variables), which disallow us from easily ruling out either value for a given variable.
- *What algorithmic paradigm would work well for the problem?* For example, again, DP works well on trees.
- *Why is the general problem hard in the first place?* Understanding this could make it easier to see why this specific case is tractable.

III. Example 1 – Size of the Largest Independent Set of a Tree

Input: A graph G that's constrained to be a tree
Output: The size of the largest independent set in G

What's hard about the original independent set problem? One answer would be that because we know nothing about the graph's structure in advance, we can only add one node at a time based on some general heuristic and, at the end, check the number of nodes in our independent set. There are exponentially many

such independent sets, and we don't know of any heuristic that's significantly better than randomly adding nodes until we can fit no more.

Here, our graph is a tree; it thus has a hierarchical structure, and we can thus consider nodes one layer at a time, starting from the bottom. This suggests a dynamic programming solution in which each node (or its corresponding subtree) has an associated subproblem.

When inspecting a node p, we should consider the following:

- If we add p, what nodes are ruled out? The parent and children of p are ruled out. Thus, this suggests that the subproblem for a node p should keep track of the max-sized independent up to p, assuming p is *absent* from the set. We will refer to these subproblems as $a(p)$.
- If we don't add p, what nodes are in? The parent and children of p are in. This suggests that the subproblem for a node p should also keep track of the max-sized independent up to p, assuming p is *contained* in the set. We will refer to these subproblems as $c(p)$.

We now have our subproblems.

Subproblems: For a given node p, keep track of $a(p)$ and $c(p)$ as described above.

Base Case: For any leaf nodes l, $\boldsymbol{a(l) = 0}$ and $\boldsymbol{c(l) = 1}$.

Iteration: Work our way up the tree. Let p be our current node and let $c_1, \dots, c_m$ be its children. Then, our subproblems are as follows:

- $\boldsymbol{a(p) = \sum_i c_i}$. Since p is not in the set, we simply add up the sets of the subtrees.
- $\boldsymbol{c(p) = 1 + \sum_i a_i}$. Since p is in the set, its children are out.

Solution: Let r be the root of the tree. Either r is in the largest independent set, or it isn't; therefore, our solution is $\boldsymbol{\max\{a(r), c(r)\}}$.

Correctness by Induction

Base Case: At leaf nodes, the subtree consists of only one node. Thus, if the node is in, the independent set has size 1; if not, it has size 0. Our base case covers this.

Induction Hypothesis: Suppose all $a(p), c(p)$ are correctly computed up to an arbitrary layer k.

Induction Step: We would like to prove that if the hypothesis holds, then $a(p), c(p)$ are correctly computed for layer $k + 1$. Let $\bar{p}$ be an arbitrary node on this layer.

- If $\bar{p}$ is not in the set, then because we want the largest independent set, we should include all of its children. By the hypothesis, all of its children have had $c(p)$ correctly computed for their subtrees, so $a(\bar{p})$ is in turn correctly computed.
- If $\bar{p}$ is in the set, then we add 1 to account for its inclusion; however, now we can't include any of its children. By the hypothesis, all of its children have had $a(p)$ correctly computed for their subtrees, so $c(\bar{p})$ is in correctly computed.

By our induction step, the "iteration" part of our program is correct.

Overall, our base case, hypothesis, and step are correct, so our algorithm is correct.

IV. Approximations of Intractable Problems – Questions to Ask

Given a problem, an algorithm, and an error bound, we are asked to prove the bound. There are two general things to consider. The following two bullet points will assume that our problem is one of selecting items to maximize some quantity; please adapt them accordingly if you're looking at a minimization problem.

- *What is the maximum possible value of the optimum, OPT?* For example, an independent set for a graph with $|V|$ nodes can obviously contain no more than $|V|$ nodes.
- Suppose you're adding items one at a time and, for each one, you preclude some other items from being added (for simplicity, assume that all items are worth the same). *At most, how many items do you forfeit for each one you add?* For example, if you forfeit at most one item for each one added, then you can argue that your algorithm will obtain at least *half* of the optimum.

One or both of these questions may be helpful for proving a bound on a given problem.

Given only a problem and a desired error bound, one may find the above questions helpful for deriving an algorithm.

V. Example 2 – Approximation of Cardinality Max Cut

We will solve the following problem from a past exam.

Given an undirected graph $G = (V, E)$, the cardinality maximum cut problem asks for a partition of V into sets S and $\bar{S} = V - S$ so that the number of edges running between these sets is maximized. Consider the following greedy algorithm for this problem. Here v_1 and v_2 are arbitrary vertices in G, and for $A \subseteq V$, $d(v, A)$ denotes the number of edges between vertex v and the set of vertices A.

```
1  Initialization:
2      A <- {v1}
3      B <- {v2}
4  for v ∈ V - {v1, v2} do:
5      if d(v,A) ≥ d(v,B) then B <- B ∪ {v}
6      else A <- A ∪ {v}
7  Output A and B
```

Show that the above polynomial-time algorithm for this problem has approximation ratio $\frac{1}{2}$.

Begin by asking our questions from part IV.

- *What is the maximum possible value of the optimum, OPT?* The tightest bound appears to be $|E|$, as one can come up with graphs in which all edges can be in the cut. Whether this question is helpful for this problem is not immediately apparent.
- *At most, how many items do you forfeit at each step?* This might be the better question.
  - At each step, we consider a total of $d(v, A) + d(v, B)$ edges.
  - It could be that OPT contains *all* of these edges, while we can only get $\max\{d(v, A), d(v, B)\}$. Therefore, **our algorithm forfeits up to $\min\{d(v, A), d(v, B)\}$ edges that OPT has**.
  - We argue that
    $\min\{d(v, A), d(v, B)\} \leq \frac{1}{2}[d(v, A) + d(v, B)]$,
    with equality only if $d(v, A) = d(v, B)$. This can be confirmed with algebraic reasoning.
  - Thus, our algorithm misses *at most* $\frac{1}{2}$ of the edges that OPT selects and, equivalently, **selects *at least* $\frac{1}{2}$ of them**. Our proof is done.

As it turns out, the second question was all we needed for this problem.

References

1. Kleinberg, Jon, and Éva Tardos. *Algorithm Design*. Boston: Pearson/Addison-Wesley, 2006. Print.