CS 341: ALGORITHMS (S18) — LECTURE 22
APPROXIMATION ALGORITHMS

ERIC BLAIS

1. Solving hard optimization problems

We saw that for many natural optimization problem—the Travelling Salesman Problem (TSP), Vertex Cover, Clique, etc.—even the (simpler) decision versions of the problems are **NP**-complete, and so we believe that there are no polynomial-time algorithms that solve those problems exactly.

As we have already mentioned a few weeks ago, if we really need to solve those problems, then we are faced with an important choice. We can:

- Use a *heuristic* algorithm that might work in practice but does not have any provable guarantees;
- Solve the problem exactly, using an exhaustive search algorithm; or
- Solve the problem *approximately* in polynomial time.

Today, we will briefly touch upon this last option, to see how for many optimization problems, there are very simple algorithms that might not always return the optimal solution but (provably!) always return a solution that is "not much worse" than the optimal solution.

2. Metric TSP

In the options we listed above, we assume that you *really* want to solve the hard problem itself. But of course in practice, your first task should be to determine if there's another problem you can solve—or if it's possible to solve just a *special case* of the hard problem.

For example, TSP is used to model the minimum distance that a travelling salesman has to drive between cities to make its sales tour... but in practice we don't need to force the salesperson to visit each city *exactly* once if driving through one city on the way to another one is shorter than driving around it. So we can remove the condition, and from our original graph we can construct a new complete graph where the weight between the vertices u and v is the *shortest path* distance between u and v in the original graph. These distances will not be arbitrary, as they will now satisfy the *triangle inequality*. The problem of solving TSP on such graphs is known as the *metric TSP* problem.

Definition 22.1. In the MetricTSP problem, we are given a complete weighted graph $G = (V, E)$ with a weight function $w : E \to \mathbb{R}^{>0}$ that satisfies

$$w(u, v) \leq w(u, x) + w(x, v)$$

for every $u, v, x \in V$; our goal is to find the length ℓ_{TSP} of the shortest tour of G that visits each vertex in V.

MetricTSP is **NP**-complete, but there is a simple algorithm that is guaranteed to return a tour of G with length at most twice that of the optimal tour. Such an algorithm is known as a *2-approximation algorithm*.

Definition 22.2. For any $k \geq 1$ and any minimization problem P, a *k-approximation algorithm* is an algorithm A where for every instance x of P with optimal value OPT, the algorithm $A(x)$ outputs a solution with value at most $k \cdot \text{OPT}$.

The 2-approximation algorithm for MetricTSP is as follows.

Algorithm 1: ApproxMetricTSP$(G = (V, E), w)$

$T \leftarrow$ a MST of G;
$L \leftarrow$ list of vertices visited in an in-order traversal of T;
tour $\leftarrow$ tour through G obtained by shortcutting paths in L that visit previously-seen vertices;
return *tour*;

The algorithm simply gets a minimum spanning tree T of the graph G (using Kruskal or Prim's algorithms, for example), then visits the edges of this tree to obtain a tour through the graph that visits each vertex at least once (but possibly multiple times). The final solution returned is the tour of the graph obtained by replacing paths that revisit some of the vertices along the way with the shortest paths to the next unvisited vertex. This algorithm takes polynomial time.

Theorem 22.3. ApproxMetricTSP *is a 2-approximation algorithm for* MetricTSP.

Proof. Let ℓ_{OPT} be the length of the optimal TSP tour through G and let ℓ_{MST} be the total length of the minimum spanning tree of G. Then

$$\ell_{MST} \leq \ell_{TSP}$$

because if we delete any edge from the TSP tour of G, we obtain a spanning tree of G. Also, by using the triangle inequality and observing that the in-order traversal of T visits each edge of the tree twice, we see that the length ℓ of the tour returned by the algorithm satisfies

$$\ell \leq 2\ell_{MST}.$$

Therefore, $\ell \leq 2\ell_{MST} \leq 2\ell_{OPT}$. □

3. Vertex Cover

Here's a nice and simple greedy algorithm for the VertexCover problem.

Algorithm 2: ApproxVC$(G = (V, E))$

$C \leftarrow \emptyset$;
for *each* $(u, v) \in E$ **do**
  if $u \notin C$ *and* $v \notin C$ **then**
    $C \leftarrow C \cup \{u, v\}$;
return C;

The ApproxVC algorithm runs in polynomial time, but it's not immediately clear that it returns a good approximation of the minimum vertex cover of G. As it turns out, it does quite well: the cover C returned by the algorithm can be at most twice as large as the optimal cover C_{OPT} of G.

Theorem 22.4. ApproxVC *is a 2-approximation algorithm to the* VertexCover *problem.*

Proof. Fix any input graph G. Let $M \subseteq E$ be the set of edges (u, v) that caused the algorithm to add u and v to the cover C. Then M forms a *matching* in G: no two edges in M share a common end vertex. We have that

$$|C| = 2|M|,$$

since both endpoints of every edge in M is added to the cover C. But we also have that

$$|M| \leq |C_{OPT}|$$

since any vertex cover of G must include at least one of the endvertices of each edge in M. Therefore, $|C| = 2|M| \leq 2|C_{OPT}|$. □

It's very tempting to try to improve on the ApproxVC algorithm by including only one of the two vertices of an edge that was not previously covered by C. (Perhaps taking the vertex that has the largest degree?) You should certainly try to do so! But you should also be warned that many people have tried without success so far: if you can obtain a k-approximation algorithm for VertexCover for any constant $k < 2$, this will be a significant breakthrough in algorithms research.

4. TSP

Emboldened by our success so far, we may want to conjecture that *every* **NP**-complete optimization problem has a polynomial-time 2-approximation algorithm, or some other reasonable approximation algorithm. This would be fantastic! Unfortunately, it's not true (Unless, of course, $\mathbf{P} = \mathbf{NP}$).

Theorem 22.5. *If* $\mathbf{P} \neq \mathbf{NP}$*, then for any constant* $k \geq 1$*, there is no polynomial-time* k*-approximation algorithm for TSP.*

Proof. We prove the contrapositive statement: that if there exists some constant k for which there is a polynomial-time algorithm A that is a k-approximation algorithm for TSP, then $\mathbf{P} = \mathbf{NP}$. We obtain this conclusion by showing that such an algorithm A can be used to design a polynomial-time algorithm for the **NP**-complete problem HamCycle.

Algorithm 3: HamCycleSolver($G = (V, E)$)

```
Construct the complete graph G' = (V, (V choose 2));
for each (u, v) ∈ (V choose 2) do
    if (u, v) ∈ E then
        w(u, v) ← 1;
    else
        w(u, v) ← kn;
T ← A(G');
if length(T) ≤ kn then
    return Yes;
else
    return No;
```

If G contains a Hamiltonian cycle, then there is a tour of G' that uses only edges of weight 1 and so has total length n. This means that $A(G')$ returns a tour of length at most kn, and the algorithm HamCycleSolver correctly returns `Yes`.

If G contains no Hamiltonian cycle, then any tour of G' must use at least one edge of weight kn, so its total length must be at least $(n-1)+kn > kn$. But then any tour returned by A has length greater than kn and HamCycleSolver correctly returns `No`. □

5. Concluding remarks

These three examples provide just a tiny sample of the richness of approximation algorithms: there are problems in this area that can be computed with simple greedy algorithms; others that require complex arguments; some problems can be approximated to *arbitrary* accuracy in polynomial time; some can't be approximated to constant factors but can be approximated to $\log n$ factors; still others have no reasonable approximation algorithm whatsoever. And the idea of designing algorithms that approximately solve a given problem does not just help with overcoming **NP**-completeness—the same idea has given rise to *sublinear-time* algorithms (that only need to examine a tiny fraction of the input) and to *sublinear-space* (or *streaming*) algorithms—that only need to examine the data in a linear stream.