Algorithms & Complexity: Lecture 13: Coping with **NP**-hardness

Sam Barrett

March 23, 2021

If we assume that $\mathbf{P} \neq \mathbf{NP}$, then it follows that **no** **NP**-hard problem has a polytime solution. But we still want to be able to solve these problems in the most efficient way possible.

We will now look at designing exact exponential time algorithms for **NP**-hard problems as these are still preferable to brute force. (**note** here *exact* simply means that the algorithm solves the problem *exactly*).

1 Algorithms via branching

1.1 Example: Vertex Cover

Problem 1 *(Vertex Cover) Given an undirected graph $G = (V, E)$ on n vertices and m edges, find a set X of minimum size s.t. each edge of G has at least one endpoint in X*

The brute force approach to this problem runs in $2^n \cdot n^{O(1)}$ time and works by:

- Enumerating all 2^n subsets of vertex set V
- For each set $X \subseteq V$, check in $O(|X|)$ time if each of the m edges has at least one endpoint in X

We now ask ourselves: Can we design an *"Can we design an $(2-\epsilon)^n \cdot n^{O(1)}$"* time algorithms for some $\epsilon > 0$?

Note we focus on minimising the 2 in $2^n \cdot n^{O(1)}$ as:

$$\lim_{n\to\infty} \left(2^n > \forall k \in \mathbb{N}.n^k\right)$$

Or as n gets increasingly larger, the 2^n gets *exponentially* larger.

Our algorithm relies on the observation that there is no point adding any vertex of degree 1 to the vertex cover. I.e. a vertex only connects to one other vertex. In this case we lose nothing by just adding the vertex it is connected

to. This second vertex has the possibility of being as good **or better** than the degree-1 vertex.

Now we can assume that there will be **no vertices of degrees < 2**. We can then say that for each vertex v of degree ≥ 2

- If we pick v then the number of vertices (remaining to be *covered*) reduces by 1
- If we do not pick v then the number of vertices reduces by at least $2+1 = 3$. This is as by not picking v **all** of its neighbours must now be picked and v has at least 2 neighbours. (2 refers to the (at least) 2 neighbours of v and the 1 refers to v being implicitly covered by selecting its neighbours)

If we define $T(n)$ as the time needed to solve the `VertexCover` problem for a graph with n vertices, we can construct the following recurrence:

$$T(n) \leq T(n-1) + T(n-3)$$

Where we solve both instances: where we pick v and where we do not and take the minimum of the two results. This gives us a total running time of $T(n-1) + T(n-3)$

Base case:

$T(2) = 1$ as for the case where the graph is two vertices connected by a single edge we select one node and have the minimum vertex cover. This requires a single operation.

Our recurrence is what is known as a **linear homogenous recurrence**

- *linear* - terms on the RHS are raised to the power 1
- *homogenous* - there are no constants on the RHS

The standard method for solving such a recurrence is to set $T(n) = x^n$ and solve for x:

- $T(n) \leq T(n-1) + T(n-3) \mapsto x^n \leq x^{n-1} + x^{n-3}$
- Taking x^{n-3} as a common factor gives: $x^3 \leq x^2 + 1$
- This is satisfied by $\forall x \leq 1.46557$
- Therefore, we can say that $1.47^n \cdot n^{O(1)}$ is an upper bound for our branching algorithm.

2 Algorithms via Dynamic Programming

2.1 Example: Travelling salesman problem (TPS)

Problem 2 *(Travelling Salesman Problem)*

- *Given a set C of n cites $c_1, c_2, \ldots, c_n$*
- *A distance function $\texttt{dist} : C \times C \to \mathbb{R}^{\geq 0}$ which gives the distance between every pair of cities*
- *We want to start at c_1 and end at c_1 after visiting **all** cities on the way*
- *What order should we visit each city to minimise the total distance we have travelled?*

The brute force approach tries all $n!$ orderings. And for each computes its cost by summing all n values. Hence, the running time of this approach is $n! \approx \left(\frac{n}{e}\right)^n$. (Where e is the Euler constant equal to around 2.718)

2.1.1 Deriving our recurrence relation

For each $S \subseteq \{c_2, c_3, \ldots, c_n\}$ and each $c_i \in S$, let $\texttt{OPT}[S, c_i]$ be the minimum length of a tour that starts at c_1, visits all cities in S and ends at c_i

Base Case: when $|S| = 1$; for each $i \geq 2$ we have $\texttt{OPT}[\{c_i\}, c_i] = \texttt{dist}(c_1, c_i)$

Now suppose that $|S| > 1$ and we want to compute $\texttt{OPT}[S, c_i]$ for some $c_i \in S$.

- Let c_j be the last city visited before ending at c_i
- c_j **must** be in $S \backslash \{c_i\}$
- Looking at all possibilities gives the following recurrence:

$$\texttt{OPT}[S, c_i] = \min_{c_j \in (S \backslash \{c_i\})} \texttt{OPT}[S \backslash \{c_i\}, c_j] + \texttt{dist}(c_j, c_i)$$

Using this recurrence we can construct the algorithm **??** and analyse its running time:

The number of entries in our table $\texttt{OPT}$ is $O(2^n \cdot n)$ as we maintain an entry $\texttt{OPT}[S, c_i]$ for each $S \subseteq \{c_2, \ldots, c_n\}$ and each $c_i \in S$

To compute $\texttt{OPT}[S, c_i]$ we take the minimum of $|S|$ numbers. To do this we look up $|S| - 1$ entries and do $|S| - 1$ addition operations. Giving an overall running time on $O(n)$ since $|S| \leq n - 1$

Algorithm 1: Travelling Salesman Problem (TSP)

Input: A set $C = \{c_1, c_2, \ldots, c_n\}$ of n cities and a distance function $\texttt{dist} : C \times C \to \mathbb{R}^{\geq 0}$

Output: The value/distance of a tour which minimises the total distance travelled when starting and ending at c_1, while visiting all cities from C

```
for i = 2 : n do
    OPT[{c_i}, c_i] = dist(c_1, c_i)
end
for k = 2 : n do
    for S ⊆ {c_2, c_3, ..., c_{n-1}, c_n} with |S| = k do
        OPT[S, c_i] = min_{c_j in (S\{c_i})} OPT[S\{c_i}, c_j] + dist(c_j, c_i)
    end
    k++
end
return min_{2≤i≤n} OPT[{c_2, c_3, ..., c_{n--1}, c_n}, c_i] + dist(c_i, c_1)
```

2.2 Set Cover problem

Problem 3 *(Set Cover problem) Given:*

- *A set $U = \{u_1, u_2, \ldots, u_N\}$ of N elements*
- *A set $\mathcal{S} = \{S_1, S_2, \ldots, S_M\}$ of non-empty subsets of U s.t. $|\mathcal{S}| = M$*

Find a collection $\mathcal{S}'$ from $\mathcal{S}$ of minimum size s.t. the unions of sets in $\mathcal{S}'$ covers all elements of U. We assume that the union of all sets in $\mathcal{S}$ covers all elements in U

The brute force approach to this problem takes $O(2^M \cdot M \cdot N)$ time as it:

- Tries all possible 2^M subsets of $\mathcal{S}$
- On each subset we can check in $O(M \cdot N)$ time if the union of all sets in the subset covers all elements in U.

  Each of the $\leq M$ sets in our subset $\mathcal{S}'$ can have at most N elements each.

Can we do better than this?

2.2.1 Setting up our recurrence

For each non-empty subset $X \subseteq U$ and each $1 \leq j \leq M$, let $\texttt{OPT}[X, j]$ be the size of the minimum cardinality subset of $\{S_1, S_2, \ldots, S_j\}$ that covers all elements from X

Base Case: $j = 1$

TODO: complete this along with formative assignment (6?)