Advanced Algorithms

Exact algorithms for NP-hard problems

Traveling Salesman Problem and Maximal Independent Set

Diana Sieper · WS22
Examples of NP-hard problems

Many important (practical) problems are NP-hard, for example . . .

- TSP
- MIS
- Bin Packing
- Scheduling
- SAT
- Graph Drawing
- Games

\[(x_1 \lor x_2 \lor \neg x_4) \land
(x_3 \lor x_7 \lor \neg x_8) \land
\ldots\]
Formal view on NP-hardness

But what does NP-hard/-complete actually mean?

- **NP-hard =** non-deterministic polynomial-time hard

- A decision problem $H$ is NP-hard when it is “at least as hard as the hardest problems in NP”.

- or: There is a polynomial-time many-one reduction from an NP-hard problem $L$ to $H$.

- If $P \neq NP$, then NP-hard problems cannot be solved in polynomial time.
Misconceptions about NP-hardness

Common misconceptions [Mann ’17]

■ If similar problems are NP-hard, then the problem at hand is also NP-hard.

■ Problems that are hard to solve in practice by an engineer are NP-hard.

■ NP-hard problems cannot be solved optimally.

■ NP-hard problems cannot be solved more efficiently than by exhaustive search.

■ For solving NP-hard problems, the only practical possibility is the use of heuristics.
Dealing with NP-hard problems

What should we do?

- Sacrifice optimality for speed
  - Heuristics (Simulated Annealing, Tabu-Search)
  - Approximation Algorithms (Christofides-Algorithm)
- Optimal Solutions
  - Exact exponential-time algorithms
  - Fine-grained analysis – parameterized algorithms
Motivation

Exponential running time ... should we just give up?

- ... can be "fast" for medium-sized instances:
  - "hidden" constants in polynomial-time algorithms:
    \[ 2^{100}n > 2^n \text{ for } n \leq 100 \]
  - \[ n^4 > 1.2^n \text{ for } n \leq 100 \]
  - TSP solvable exactly for \( n \leq 2000 \) and specialized instances with \( n \leq 85900 \)

Efficient (polynomial-time) vs. inefficient (super-pol.time)
Motivation

Exponential runningtime . . . maybe we need better hardware?

- Suppose an algorithm uses $a^n$ steps & can solve for a fixed amount of time $t$ instances up to size $n_0$.
- Improving hardware by a constant factor $c$ only adds a constant (relative to $c$) to $n_0$:
  \[ a^{n'_0} = c \cdot a^{n_0} \quad \leadsto \quad n'_0 = \log_a c + n_0 \]
- Reducing the base of the runtime to $b < a$ results in a multiplicative increase:
  \[ b^{n'_0} = a^{n_0} \quad \leadsto \quad n'_0 = n_0 \cdot \log_b a \]
Motivation

Exponential runningtime . . . but can we at least find exact algorithms that are faster than **brute-force** (trivial) approaches?

- **TSP**: Bellman-Held-Karp algorithm has running time $O(2^n n^2)$ compared to an $O(n! \cdot n)$-time brute-force search.

- **MIS**: algorithm by Tarjan & Trojanowski runs in $O(2^{n/3})$ time compared to a trivial $O(n2^n)$-time approach.

- **Coloring**: Lawler gaven an $O(n(1 + 3\sqrt{3})^n)$ algorithm compared to $O(n^{n+1})$-time brute-force.

- **SAT**: No better algorithm than trivial brute-force search known.
$\mathcal{O}^*$-notation

$\mathcal{O}(1.4^n \cdot n^2) \subsetneq \mathcal{O}(1.5^n \cdot n) \subsetneq \mathcal{O}(2^n)$

- base of exponential part dominates $\rightsquigarrow$ negligible polynomial factors

$\forall f(n) \in \mathcal{O}^*(g(n)) \iff \exists$ polynomial $p(n)$ with $f(n) \in \mathcal{O}(g(n)p(n))$

- typical result

<table>
<thead>
<tr>
<th>Approach</th>
<th>Runtime in $\mathcal{O}$-Notation</th>
<th>$\mathcal{O}^*$-Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute-Force</td>
<td>$\mathcal{O}(2^n)$</td>
<td>$\mathcal{O}^*(2^n)$</td>
</tr>
<tr>
<td>Algorithm A</td>
<td>$\mathcal{O}(1.5^n \cdot n)$</td>
<td>$\mathcal{O}^*(1.5^n)$</td>
</tr>
<tr>
<td>Algorithm B</td>
<td>$\mathcal{O}(1.4^n \cdot n^2)$</td>
<td>$\mathcal{O}^*(1.4^n)$</td>
</tr>
</tbody>
</table>
Traveling Salesperson Problem (TSP)

**Input.** Distinct cities \( \{v_1, v_2, \ldots, v_n\} \) with distances \( d(c_i, c_j) \in \mathbb{Q}_{\geq 0} \); directed, complete graph \( G \) with edge weights \( d \)

**Output.** Tour of the traveling salesperson of minimal total length that visits all the cities and returns to the starting point;

i.e. a Hamiltonian cycle \( (v_{\pi(1)}, \ldots, v_{\pi(n)}, v_{\pi(1)}) \) of \( G \) of minimum weight

\[
\sum_{i=1}^{n-1} d(v_{\pi(i)}, v_{\pi(i+1)}) + d(v_{\pi(n)}, v_{\pi(1)})
\]

**Brute-force.**

- Try all permutations and pick the one with smallest weight.
- Runtime: \( \Theta(n! \cdot n) = n \cdot 2^{\Theta(n \log n)} \)
TSP – Dynamic programming

Bellman-Held-Karp algorithm

Idea.

- Reuse optimal substructures with dynamic programming.
- Select a starting vertex $s \in V$.
- For each $S \subseteq V - s$ and $v \in S$, let:
  $$OPT[S, v] = \text{length of a shortest } s-v\text{-path that visits precisely the vertices of } S \cup \{s\}.$$


Bellman-Held-Karp algorithm
TSP – Dynamic programming

Details.

- The base case $S = \{v\}$ is easy: $\text{OPT}[\{v\}, v] = d(s, v)$.
- When $|S| \geq 2$, compute $\text{OPT}[S, v]$ recursively:

$$\text{OPT}[S, v] = \min \{ \text{OPT}[S - v, u] + d(u, v) \mid u \in S - v \}$$

- After computing $\text{OPT}[S, v]$ for each $S \subseteq V - s$ and each $v \in V - s$, the optimal solution is easily obtained as follows:

$$\text{OPT} = \min \{ \text{OPT}[V - s, v] \} + d(v, s) \mid v \in V - s \}$$
TSP – Dynamic programming

**Pseudocode.**

Algorithm Bellmann-Held-Karp\((G, c)\)

```plaintext
foreach \( v \in V - s \) do
    \( \text{OPT}[\{v\}, v] = c(s, v) \)

for \( j \leftarrow 2 \) to \( n - 1 \) do
    foreach \( S \subseteq V - s \) with \( |S| = j \) do
        foreach \( v \in S \) do
            \( \text{OPT}[S, v] \leftarrow \min\{ \text{OPT}[S - v, u] + c(u, v) \mid u \in S - v \} \)

return \( \min\{ \text{OPT}[V - s, v] + c(v, s) \mid v \in V - s \} \)
```

**Analysis.**

- innermost loop executes \( \mathcal{O}(2^n \cdot n) \) iterations
- each takes \( \mathcal{O}(n) \) time
- total of \( \mathcal{O}(2^n n^2) = \mathcal{O}^*(2^n) \)
- Space usage in \( \Theta(2^n \cdot n) \)
- Or actually better? What table values do we need to store?

■ A shortest tour can be produced by backtracking the DP table (as usual).
TSP – Discussion

- DP algorithm that runs in $O^*(2^n)$ time and $O(2^n \cdot n)$ space
- Brute-force runs in $2^{O(n \log n)}$ time
  ⇒ Sacrifice space for speedup
- Many variants of TSP: symmetric, asymmetric, metric, vehicle routing problems, . . .
- Metric TSP can easily be 2-approximated. (Do you remember how?)
- Euclidean TSP is considered in the course Approximation Algorithms.

- In practice, one successful approach is to start with a greedily computed Hamiltonian cycle and then use 2-OPT and 3-OPT swaps to improve it.
Maximum Independent Set (MIS)

**Input.** Graph $G = (V, E)$ with $n$ vertices.

**Output.** Maximum size independent set, i.e., a largest set $U \subseteq V$, such that no pair of vertices in $U$ are adjacent in $G$.

**Brute-force.**
- Try all subsets of $V$.
- Runtime: $O(2^n \cdot n)$

**Naive MIS branching.**
- Take a vertex $v$ or don’t take it.

Algorithm NaiveMIS($G$)

```
if $V = \emptyset$ then
  return 0

$v \leftarrow$ arbitrary vertex in $V(G)$

return max\{1+ NaiveMIS($G - N(v) - \{v\}$),
            NaiveMIS($G - \{v\}$)\}
$3 + 1 = 4$

$1 + 1 = 2$

$3 + 2 = 5$

$1 + 0 = 1$
MIS – Smarter branching

Lemma.
Let $U$ be a maximum independent set in $G$. Then for each $v \in V$:
1. $v \in U \Rightarrow N(v) \cap U = \emptyset$
2. $v \notin U \Rightarrow |N(v) \cap U| \geq 1$
Thus, $N[v] := N(v) \cup \{v\}$ contains some $y \in U$ and no other vertex of $N[y]$ is in $U$.

Smarter MIS branching.

■ For some vertex $v$, branch on vertices in $N[v]$.

Algorithm MIS($G$)

if $V = \emptyset$ then
  return 0
$v \leftarrow$ vertex of minimum degree in $V(G)$
return $1 + \max\{\text{MIS}(G - N[y]) \mid y \in N[v]\}$

■ Correctness follows from Lemma.
■ We prove a runtime of $O^*(3^n/3) = O^*(1.4423^n)$. 

Correctness follows from

\[
O^*(3^n/3) = O^*(1.4423^n).
\]
MIS – Branching analysis

Execution corresponds to a search tree whose vertices are labeled with the input of the respective recursive call.

- Let $B(n)$ be the maximum number of leaves of a search tree for a graph with $n$ vertices.
- Search-tree has height $\leq n$.

$\Rightarrow$ The algorithm’s runtime is

\[
T(n) \in O^*(nB(n)) = O^*(B(n)).
\]

- Let’s consider an example run.
MIS – Runtime analysis

For a worst-case $n$-vertex graph $G \ (n \geq 1)$:

$$B(n) \leq \sum_{y \in N[v]} B(n - (\deg(y) + 1)) \leq (\deg(v) + 1) \cdot B(\ n - (\deg(v) + 1) )$$

where $v$ is a minimum degree vertex of $G$, and we note that $B(n') \leq B(n)$ for any $n' \leq n$.

We prove by induction that $B(n) \leq 3^{n/3}$.

- **Base case:** $B(0) = 1 \leq 3^{0/3}$
- **Hypothesis:** for $n \geq 1$, set $s = \deg(v) + 1$ in the above inequality

$$B(n) \leq s \cdot B(n - s) \leq s \cdot 3^{(n-s)/3} = \frac{s}{3^{s/3}} \cdot 3^{n/3} \leq 3^{n/3}$$

$$B(n) \in O^*(\sqrt[3]{n}) \subset O^*(1.44225^n)$$
MIS – Discussion

- Smarter branching leads to $O^*(1.44225^n)$-time algorithm,
- compared to brute-force, which runs in $O^*(2^n)$ time.

- Algorithms for MIS known that run in $O^*(1.2202^n)$ time and polynomial space,
- and in $O^*(1.2109^n)$ time and exponential space.

- What vertices are always in a MIS?
- What vertices can we safely assume are in a MIS?
- Advanced case analysis in [Fomin, Kratsch Ch 2.3] leading to a $O^*(1.2786^n)$-time algorithm.

- Exercise: Edge-branching for MIS
Literature

Main source:
- [Fomin, Kratsch Ch1] “Exact Exponential Algorithms”

Referenced papers:
- [ADMV '15] Classic Nintendo Games are (Computationally) Hard
- [Mann '17] The Top Eight Misconceptions about NP-Hardness