Gems of TCS

Exponential-time Algorithms

Sasha Golovnev

September 15, 2021

Exact Algorithms

- We need to solve problem exactly

Exact Algorithms

- We need to solve problem exactly
- Problem takes exponential time solve exactly

Exact Algorithms

- We need to solve problem exactly
- Problem takes exponential time solve exactly
- Intelligent exhaustive search: finding optimal solution without going through all candidate solutions

Running Time

running time:	n	n^2	n^3	$n!$
less than 10^9:	10^9	$10^{4.5}$	10^3	12

Running Time

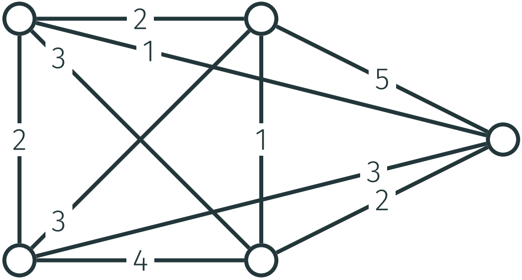
running time:	n	n^2	n^3	$n!$
less than 10^9:	10^9	$10^{4.5}$	10^3	12

running time:	$n!$	4^n	2^n	1.308^n
less than 10^9:	12	14	29	77

Traveling Salesman Problem (TSP)

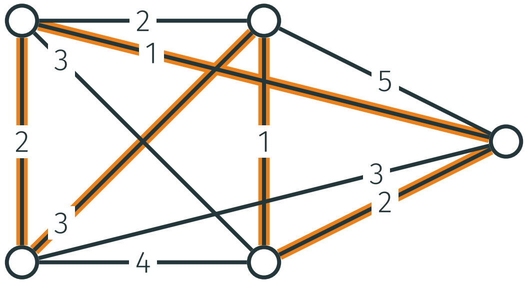
Traveling Salesman Problem

Given a complete weighted graph, find a cycle (or a path) of minimum total weight (length) visiting each node exactly once



Traveling Salesman Problem

Given a complete weighted graph, find a cycle (or a path) of minimum total weight (length) visiting each node exactly once



length: 9

Algorithms

- Classical optimization problem with countless number of real life applications (see Lecture 1)

Algorithms

- Classical optimization problem with countless number of real life applications (see Lecture 1)
- No polynomial time algorithms known

Algorithms

- Classical optimization problem with countless number of real life applications (see Lecture 1)
- No polynomial time algorithms known
- We'll see exact exponential-time algorithms

Brute Force Solution

A naive algorithm just checks all possible $\sim n!$ cycles.

Brute Force Solution

A naive algorithm just checks all possible $\sim n!$ cycles.

We'll see

- Use dynamic programming to solve TSP in $O(n^2 \cdot 2^n)$

Brute Force Solution

A naive algorithm just checks all possible $\sim n!$ cycles.

We'll see

- Use dynamic programming to solve TSP in $O(n^2 \cdot 2^n)$
- The running time is exponential, but is much better than $n!$

Dynamic Programming

- Dynamic programming is one of the most powerful algorithmic techniques

Dynamic Programming

- Dynamic programming is one of the most powerful algorithmic techniques
- Rough idea: express a solution for a problem through solutions for smaller subproblems

Dynamic Programming

- Dynamic programming is one of the most powerful algorithmic techniques
- Rough idea: express a solution for a problem through solutions for smaller subproblems
- Solve subproblems one by one. Store solutions to subproblems in a table to avoid recomputing the same thing again

Subproblems

- For a subset of vertices $S \subseteq \{1, \ldots, n\}$ containing the vertex 1 and a vertex $i \in S$, let $C(S, i)$ be the length of the shortest path that starts at 1, ends at i and visits all vertices from S exactly once

Subproblems

- For a subset of vertices $S \subseteq \{1, \ldots, n\}$ containing the vertex 1 and a vertex $i \in S$, let $C(S, i)$ be the length of the shortest path that starts at 1, ends at i and visits all vertices from S exactly once
- $C(\{1\}, 1) = 0$ and $C(S, 1) = +\infty$ when $|S| > 1$

Recurrence Relation

- Consider the second-to-last vertex j on the required shortest path from 1 to i visiting all vertices from S

Recurrence Relation

- Consider the second-to-last vertex j on the required shortest path from 1 to i visiting all vertices from S
- The subpath from 1 to j is the shortest one visiting all vertices from $S - \{i\}$ exactly once

Recurrence Relation

- Consider the second-to-last vertex j on the required shortest path from 1 to i visiting all vertices from S
- The subpath from 1 to j is the shortest one visiting all vertices from $S - \{i\}$ exactly once
- Hence $C(S, i) = \min_j \{C(S - \{i\}, j) + d_{ji}\}$, where the minimum is over all $j \in S$ such that $j \neq i$

Order of Subproblems

- Need to process all subsets $S \subseteq \{1, \ldots, n\}$ in an order that guarantees that when computing the value of $C(S, i)$, the values of $C(S - \{i\}, j)$ have already been computed

Order of Subproblems

- Need to process all subsets $S \subseteq \{1, \ldots, n\}$ in an order that guarantees that when computing the value of $C(S, i)$, the values of $C(S - \{i\}, j)$ have already been computed
- For example, we can process subsets in order of increasing size

Algorithm

$C(*, *) \leftarrow +\infty$

$C(\{1\}, 1) \leftarrow 0$

Algorithm

$C(*, *) \leftarrow +\infty$

$C(\{1\}, 1) \leftarrow 0$

for s from 2 to n:

for all $1 \in S \subseteq \{1, \ldots, n\}$ of size s:

Algorithm

$C(*, *) \leftarrow +\infty$

$C(\{1\}, 1) \leftarrow 0$

for s from 2 to n:

  for all $1 \in S \subseteq \{1, \ldots, n\}$ of size s:

    for all $i \in S$, $i \neq 1$:

      for all $j \in S$, $j \neq i$

        $C(S, i) \leftarrow \min\{C(S, i), C(S - \{i\}, j) + d_{ji}\}$

Algorithm

$C(*, *) \leftarrow +\infty$

$C(\{1\}, 1) \leftarrow 0$

for s from 2 to n:

  for all $1 \in S \subseteq \{1, \ldots, n\}$ of size s:

    for all $i \in S$, $i \neq 1$:

      for all $j \in S$, $j \neq i$

        $C(S, i) \leftarrow \min\{C(S, i), C(S - \{i\}, j) + d_{ji}\}$

return $\min_i\{C(\{1, \ldots, n\}, i) + d_{i,1}\}$

Satisfiability Problem (SAT)

SAT

$$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (x_2 \vee \neg x_3)$$

SAT

$$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (x_2 \vee \neg x_3)$$

$$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$$

k-SAT

$$
\begin{aligned}
\phi(x_1, \ldots, x_n) = & (x_1 \lor \neg x_2 \lor \ldots \lor x_k) \quad \land \\
& \qquad\qquad \ldots \qquad\qquad\qquad \land \\
& (x_2 \lor \neg x_3 \lor \ldots \lor x_8)
\end{aligned}
$$

k-SAT

$$
\begin{aligned}
\phi(x_1, \ldots, x_n) = & (x_1 \vee \neg x_2 \vee \ldots \vee x_k) \quad \wedge \\
& \qquad\qquad \ldots \qquad\qquad\quad \wedge \\
& (x_2 \vee \neg x_3 \vee \ldots \vee x_8)
\end{aligned}
$$

ϕ is satisfiable if

$$\exists x \in \{0,1\}^n \colon \phi(x) = 1\,.$$

Otherwise, ϕ is unsatisfiable

k-SAT

$$\begin{aligned}\phi(x_1,\ldots,x_n) = &(x_1 \vee \neg x_2 \vee \ldots \vee x_k) \quad \wedge \\ &\qquad\qquad \ldots \qquad\qquad\quad \wedge \\ &(x_2 \vee \neg x_3 \vee \ldots \vee x_8)\end{aligned}$$

ϕ is satisfiable if

$$\exists x \in \{0,1\}^n \colon \phi(x) = 1\,.$$

Otherwise, ϕ is unsatisfiable

n Boolean vars, m clauses

k-SAT

$$\begin{aligned}\phi(x_1, \ldots, x_n) = &(x_1 \lor \neg x_2 \lor \ldots \lor x_k) \quad \land \\ &\ldots \quad \land \\ &(x_2 \lor \neg x_3 \lor \ldots \lor x_8)\end{aligned}$$

ϕ is satisfiable if

$$\exists x \in \{0,1\}^n \colon \phi(x) = 1 .$$

Otherwise, ϕ is unsatisfiable

n Boolean vars, m clauses

k-SAT is SAT where clause length $\leq k$

k-SAT. Examples

$$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (x_2 \vee \neg x_3)$$

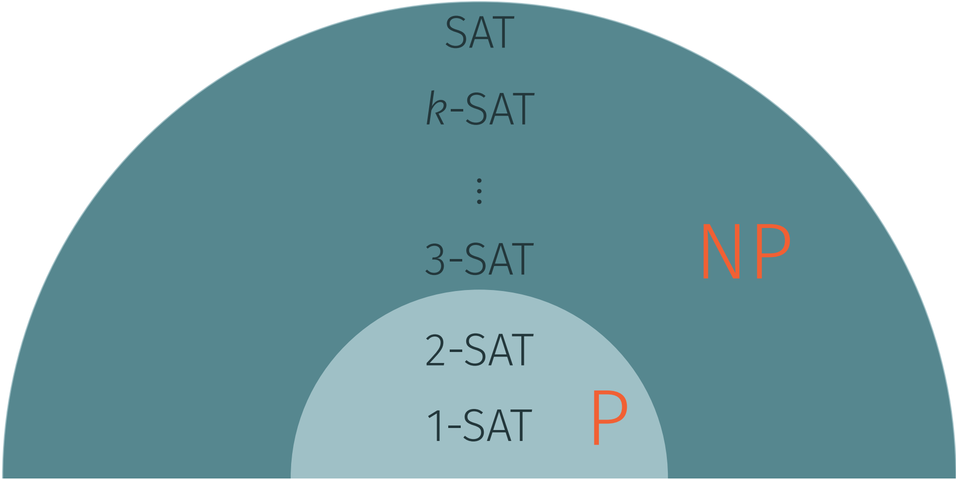
k-SAT. Examples

$$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (x_2 \vee \neg x_3)$$

$$(x_1) \wedge (\neg x_2) \wedge (x_3) \wedge (\neg x_1)$$

Complexity of SAT

Complexity of SAT



But how hard is SAT?

SAT in 2^n

- $O^*(\cdot)$ suppresses polynomial factors in the input length:

$$2^n n^{10} m^2 = O^*(2^n)$$

SAT in 2^n

- $O^*(\cdot)$ suppresses polynomial factors in the input length:

$$2^n n^{10} m^2 = O^*(2^n)$$

- SAT can be solved in time $O^*(2^n)$

SAT IN 2^n

- $O^*(\cdot)$ suppresses polynomial factors in the input length:

$$2^n n^{10} m^2 = O^*(2^n)$$

- SAT can be solved in time $O^*(2^n)$
- We don't know how to solve SAT exponentially faster: in time $O^*(1.999^n)$

3-SAT

- $(x_1 \vee x_2 \vee x_9) \wedge \ldots \wedge (x_2 \vee \neg x_3 \vee x_8)$

3-SAT

- $(x_1 \vee x_2 \vee x_9) \wedge \ldots \wedge (x_2 \vee \neg x_3 \vee x_8)$

3-SAT

- $(x_1 \vee x_2 \vee x_9) \wedge \ldots \wedge (x_2 \vee \neg x_3 \vee x_8)$
- Consider three sub-problems:
  - $x_1 = 1$
  - $x_1 = 0, x_2 = 1$
  - $x_1 = 0, x_2 = 0, x_9 = 1$

3-SAT

- $(x_1 \vee x_2 \vee x_9) \wedge \ldots \wedge (x_2 \vee \neg x_3 \vee x_8)$
- Consider three sub-problems:
    - $x_1 = 1$
    - $x_1 = 0, x_2 = 1$
    - $x_1 = 0, x_2 = 0, x_9 = 1$
- The original formula is SAT iff at least one of these formulas is SAT

3-SAT. Analysis

- $T(n) \leq T(n-1) + T(n-2) + T(n-3)$

3-SAT. Analysis

- $T(n) \leq T(n-1) + T(n-2) + T(n-3)$
- $T(n) \leq 1.85^n$:

3-SAT. Analysis

- $T(n) \leq T(n-1) + T(n-2) + T(n-3)$
- $T(n) \leq 1.85^n$:

$$
\begin{aligned}
T(n) &\leq T(n-1) + T(n-2) + T(n-3) \\
&\leq 1.85^{n-1} + 1.85^{n-2} + 1.85^{n-3} \\
&= 1.85^n\left(\frac{1}{1.85} + \frac{1}{1.85^2} + \frac{1}{1.85^3}\right) \\
&< 1.85^n(0.991) \\
&< 1.85^n
\end{aligned}
$$

3-SAT. Analysis

- $T(n) \leq T(n-1) + T(n-2) + T(n-3)$
- $T(n) \leq 1.85^n$:

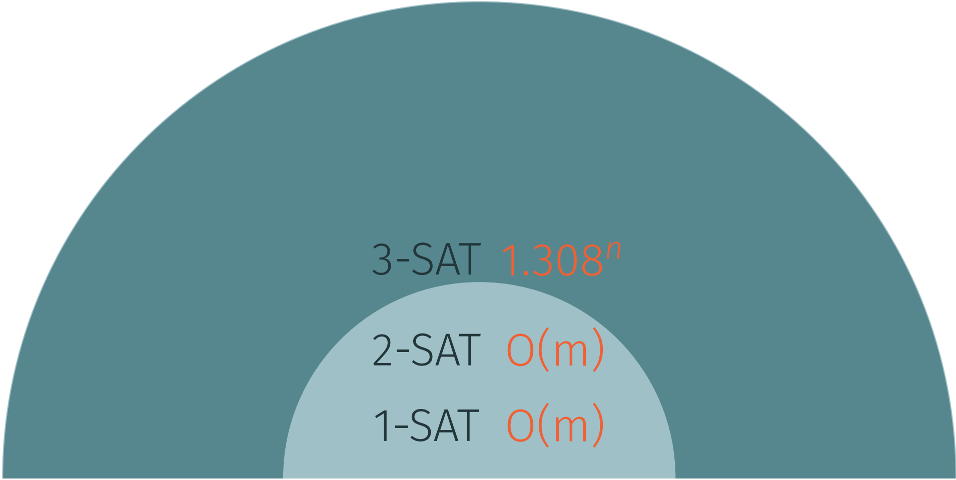
$$\begin{aligned} T(n) &\leq T(n-1) + T(n-2) + T(n-3) \\ &\leq 1.85^{n-1} + 1.85^{n-2} + 1.85^{n-3} \\ &= 1.85^n\left(\frac{1}{1.85} + \frac{1}{1.85^2} + \frac{1}{1.85^3}\right) \\ &< 1.85^n(0.991) \\ &< 1.85^n \end{aligned}$$

- There are even faster algorithms: 1.308^n [HKZZ19]

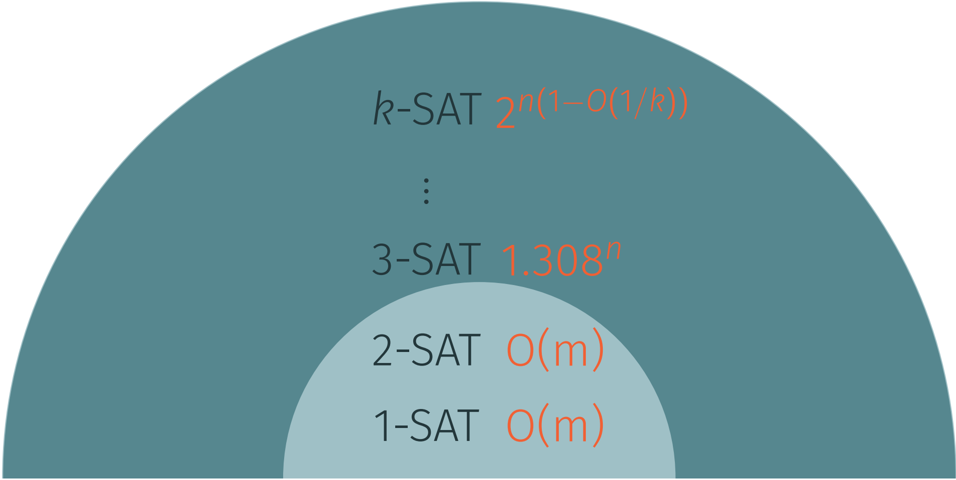
How hard can SAT be?

Algorithmic Complexity of SAT

Algorithmic Complexity of SAT



Algorithmic Complexity of SAT



Algorithmic Complexity of SAT