Computationally Hard Problems

Further Proofs of $\mathcal{NP}$-Completeness

Carsten Witt

Institut for Matematik og Computer Science
Danmarks Tekniske Universitet

Fall 2020

Hamilton Cycle/Path

An undirected graph $G = (V, E)$, $|V| = n$ has a *Hamilton Cycle* (also called *Hamilton(ian) Circuit*) if there is a permutation $v_{i_1}, v_{i_2} \ldots v_{i_n}$ of the nodes such that $\{v_{i_j}, v_{i_{j+1}}\} \in E$ for $j = 1, \ldots, n-1$, and $\{v_{i_n}, v_{i_1}\} \in E$.

If we drop the requirement $\{v_{i_n}, v_{i_1}\} \in E$, i. e., do not return, the graph has a *Hamilton Path*.

Problem [HamiltonCycle]
Input: An undirected graph $G = (V, E)$.
Output: YES if G has a Hamilton Cycle and NO otherwise.

The problem HamiltonPath is defined analogously.

Hamilton Cycle/Path

The path/cycle problems have also directed versions.

Here one has to follow the edges in the given direction.

In addition, the edges can have weights/costs/lengths.

Then the task is to find paths/cycles with minimum weight.

- Directed Hamilton path/cycle.
- Weighted directed Hamilton path/cycle.

An appropriate base problem for $\mathcal{NP}$-completeness proofs is again 3-SAT.

DIRECTEDHAMILTONPATH can be reduced to GLASSESINCUPBOARD.

$\mathcal{NP}$-completeness of DirectedHamiltonCycle

Theorem

The problem DirectedHamiltonCycle *(DHC) is* $\mathcal{NP}$*-complete.*

Proof:

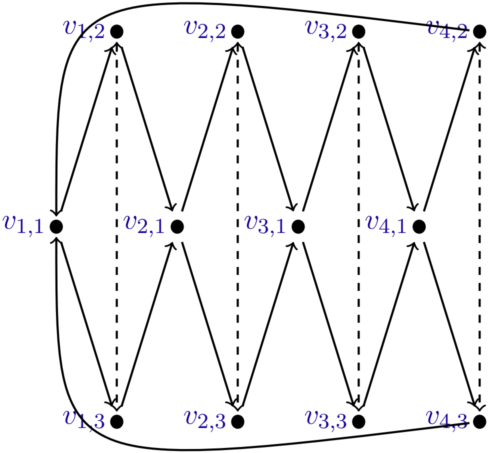
We leave the fact $\mathrm{DHC} \in \mathcal{NP}$ as an exercise.

For the poly-time reduction, the reference problem is 3-Sat, i.e., we show

$$\text{3-Sat} \leq_p \mathrm{DHC}$$

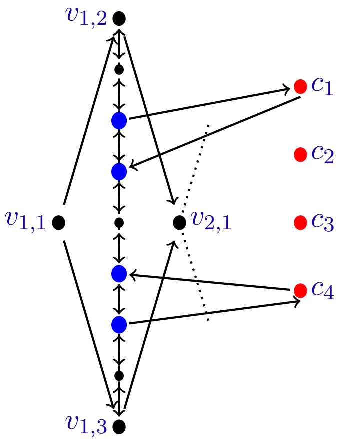
3-Sat $\leq_p$ DHC: Variable Components

Given n variables, construct a frame consisting of n components:



- Doubly-linked lists between $v_{i,2}$ and $v_{i,3}$
- Interpret direction through list as value of x_i (to $v_{i,2}$ means $x_i = 1$)
- Altogether 2^n different Hamilton cycles
- Now:
  Consider the clauses

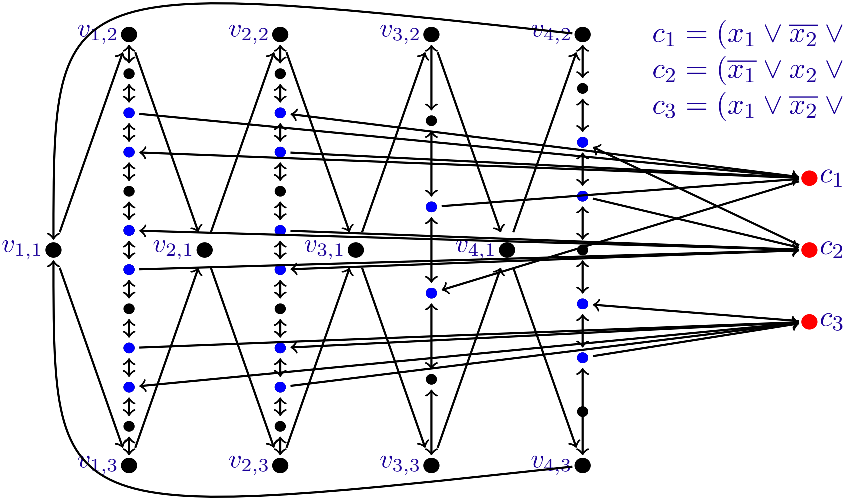
3-SAT $\leq_p$ DHC: Clause Components



- For each clause c_j, $1 \leq j \leq m$, introduce an extra node c_j.
- Will connect the c_j by breaking up the doubly-linked lists.
- Let k_i be the total number of times that x_i or $\overline{x_i}$ appears in the clauses.
- Doubly-linked list for x_i contains $2k_i$ special nodes and $k_i + 1$ helper nodes.
- If x_i appears positively in c_j, connect it from upper special node to c_j-node, otherwise from lower one.

Example: $m = 4$, $k_1 = 2$, c_1 contains x_1, c_4 contains $\bar{x}_1$.

3-SAT $\leq_p$ DHC: A Full Example



$$c_1 = (x_1 \vee \overline{x_2} \vee x_3)$$
$$c_2 = (\overline{x_1} \vee x_2 \vee \overline{x_4})$$
$$c_3 = (x_1 \vee \overline{x_2} \vee \overline{x_4})$$

3-SAT $\leq_p$ DHC: Correctness (1/2)

Show that there is a DHC if and only if there is a satisfying assignment. First assume a satisfying assignment. Construct a DHC:

- If $x_i = 1$, go from $v_{i,1}$ to $v_{i,2}$, else to $v_{i,3}$.
- Pick for each clause one literal satisfying the clause, e. g., given $c_j = (\cdot \vee \overline{x_i} \vee \cdot)$ pick $\overline{x_i}$ if $x_i = 0$.
- Visit node c_j on way from $v_{i,3}$ to $v_{i,2}$: leave doubly-linked list at lower special node towards c_j and return at upper special node. Analogously for positive literals.
- If more than one literal is satisfied in the clause, stay on doubly-linked list for the remaining satisfied literals.
- So all nodes are visited exactly once and path returns to $v_{1,1}$.
- Have constructed a DHC.

3-SAT $\leq_p$ DHC: Correctness (2/2)

Assume a DHC on the whole graph. Aim: construct a satisfying assignment. Observations:

- Each doubly-linked list must be entered from $v_{\cdot,3}$ or $v_{\cdot,2}$.
- As all helper nodes must be visited: Cycle can leave list only at special nodes towards c_j-nodes, must return at *adjacent* special node and eventually continue to end of list.
- A c_j-node can be visited on way from $v_{i,2}$ to $v_{i,3}$ only if x_i appears positively (analogously for negative appearance).

Construct assignment according to whether edge $(v_{i,1}, v_{i,2})$ or $(v_{i,1}, v_{i,3})$ is taken.

Conclude: As all c_j are visited, every clause has a literal satisfying it. Hence, the whole assignment is satisfying all clauses.

3-Sat $\leq_p$ DHC: Running Time

Easy to see: The graph can be constructed in time polynomial in the number of variables and clauses.

This completes the proof.

$\mathcal{NP}$-completeness of TRAVELINGSALESMAN

Theorem

TRAVELINGSALESMAN *is $\mathcal{NP}$-complete.*

Proof sketch: Reduction from HAMILTONCYCLE (undirected!), the $\mathcal{NP}$-completeness of which is an exercise.

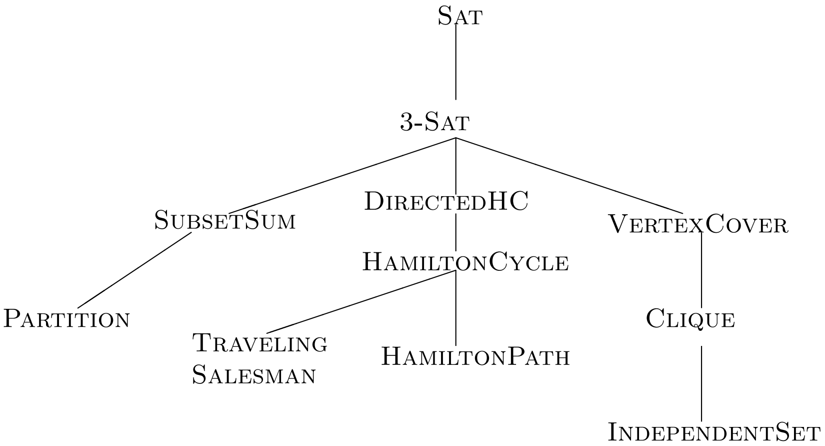
Given undirected graph $G = (V, E)$, construct distance matrix as follows:

$$C(i, j) = \begin{cases} 1 & \text{if } \{i, j\} \in E, \\ 2 & \text{otherwise.} \end{cases}$$

Ask for a tour of cost at most n. Such tours correspond one-to-one to Hamilton cycles in G.

The rest of the reasoning is as usual . . .

Overview over Reductions



Overview

There are many thousands $\mathcal{NP}$-complete problems known.

Most of them are "real world problems" that have to be – and are – solved every day, though not always optimally.

We present a number of generic problems which can often be used to show that others are hard.

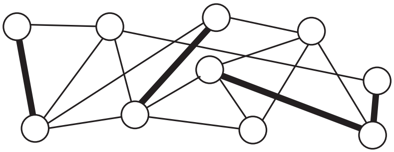
Further proofs will not be given today; some are in the lecture notes.

More Graph Problems

Problem [RURALPOSTMAN]

Input: A connected undirected graph $G = (V, E)$, weights of the edges $w(e) \in \mathbb{N}$ and a subset of required edges $E' \subseteq E$ and an bound $B \in \mathbb{N}$.

Output: YES if there is a closed walk (edges may be used more than once, not all nodes have to be visited) in G with weight at most B that traverses each edge in E' at least once; NO otherwise.



Graph Problems

Problem [MAXIMUMCUT]
Input: An undirected graph $G = (V, E)$ and a constant $k \in \mathbb{N}$
Output: YES if there is a partition of V into two sets V_1, V_2 such that there are at least k edges between V_1 and V_2. NO otherwise.

More Graph Problems

Problem [MinimumCliqueCover]
Input: An undirected graph $G = (V, E)$ and a natural number k.
Output: YES if there is clique cover for G of size at most k. That is, a collection $V_1, V_2, \ldots, V_k$ of not necessarily disjoint subsets of V such that each V_i induces a complete subgraph of G and such that for each edge $\{u, v\} \in E$ there is some V_i that contains both u and v. NO otherwise.

Partition/Selection Problems

Problem [MinimumRectangleTiling]
Input: An $n \times n$ array A of non-negative numbers, positive integers k and B.
Output: YES if there is a partition of A into k non-overlapping rectangular sub-arrays such that the sum of the entries in every sub-array is at most B. NO otherwise.

1	3	2	3	1	3
2	1	1	2	1	1
8	1	15	1	2	1
1	1	1	1	2	2
1	2	7	1	1	2
1	1	1	1	1	1

Partition/Selection Problems

Problem [BinPacking]
Input: A sequence $s_1, s_2, \ldots, s_n$ of positive rational numbers and a natural number B.
Output: YES if the objects can be packed into at most B bins and NO otherwise.

- We want to pack n objects with sizes $s_1, s_2, \ldots, s_n$ into as few bins as possible.
- Every bin has a capacity of 1.
- The objects can be be packed into the bins such that there is no space between them.
- The objects cannot be divided.

Partition/Selection Problems

Problem [KNAPSACK]
Input: Sequences $w_1, w_2, \ldots, w_n$ and $s_1, s_2, \ldots, s_n$ of natural numbers and natural numbers B, K
Interpretation: We have n objects. The i-th object has weight w_i and value s_i. We want to pack objects into a knapsack which has weight limit B such that the total value of the objects in the knapsack is maximum.
Output: YES if there is a set $A \subseteq \{1, 2, \ldots, n\}$ such that:

$$\sum_{i \in A} w_i \leq B \qquad \text{and} \qquad \sum_{i \in A} s_i \geq K \ .$$

Integer Programming

Problem [IntegerProgramming]
Input: Parameters $c_1, \ldots, c_m \in \mathbb{Z}$, $a_{j1}, \ldots, a_{jm}, b_j \in \mathbb{Z}$, $j = 1, \ldots, k$ and $B \in \mathbb{Z}$.
Output: YES if there are $x_1, \ldots, x_m \in \mathbb{Z}$ such that the following holds:

$$\begin{aligned} c_1 x_1 + \cdots + c_m x_m &\geq B \\ a_{j1} x_1 + \cdots + a_{jm} x_m &\leq b_j, \quad j = 1, \ldots, k \end{aligned}$$

and NO otherwise.

If the numbers x_i are allowed to be real numbers then the problem is efficiently solvable.

Scheduling

Problem [OneProcessorScheduling]
Input: A set T of (un-dividable) tasks. Every task $t \in T$ has a length $l(t) \in \mathbb{Z}^+$, an earliest release time $r(t) \in \mathbb{Z}_0^+$, and a deadline $d(t) \in \mathbb{Z}^+$.
Output: YES if there is a schedule σ for a single processor. That is, σ assigns a start time $\sigma(t) \in \mathbb{Z}_0^+$ to each job such that for all $t, t' \in T$

$$\begin{aligned} \sigma(t) &\geq r(t) \\ \sigma(t) + l(t) &\leq d(t) \\ \sigma(t) > \sigma(t') &\implies \sigma(t) \geq \sigma(t') + l(t') \end{aligned}$$

Scheduling

Problem [MultiProcessorScheduling]

Input: A set T of tasks, m processors and a deadline $D \in \mathbb{Z}^+$. Every task $t \in T$ has a length $l(t) \in \mathbb{Z}^+$.

Output: YES if there is a schedule σ which meets the deadline D. That is, σ assigns a start time $\sigma(t) \in \mathbb{Z}_0^+$ to each job such that

- for all times $u \in \{0, 1, \ldots, D\}$ the number of active tasks t (i. e., $\sigma(t) \leq u < \sigma(t) + l(t)$) is at most m.
- for all tasks $t \in T$ it holds that $\sigma(t) + l(t) \leq D$.

Computationally Hard Problems

Dynamic Programming and Approximation Algorithms

Carsten Witt

Institut for Matematik og Computer Science
Danmarks Tekniske Universitet

Fall 2020

A Note on Optimization Problems

For the optimization versions of some $\mathcal{NP}$-complete problems so-called approximation algorithms are known.

These are deterministic (or randomized) **polynomial-time** algorithms which guarantee to find a solution which is within a certain distance of the optimum.

The approximation ratio of such an algorithm A on problem instance $\boldsymbol{X}$ is defined by

$$R_A(\boldsymbol{X}) = \frac{A(\boldsymbol{X})}{\mathsf{OPT}(\boldsymbol{X})}\text{(minimization)};\ R_A(\boldsymbol{X}) = \frac{\mathsf{OPT}(\boldsymbol{X})}{A(\boldsymbol{X})}\text{(maximization)}$$

Sometimes $R_A(\boldsymbol{X})$ can be upper bounded in the same way for all $\boldsymbol{X}$.

For the knapsack problem we will analyze how to achieve $R_A \leq 1 + \epsilon$ (for arbitrarily small constant $\epsilon > 0$).

A Note on Optimization Problems

On the other hand, for the optimization versions of some $\mathcal{NP}$-complete problems one can show that they cannot be approximated better than some ratio.

Pseudo-Polynomial Algorithms

These are algorithms that solve the problem optimally and run in time polynomial "in some parameter of the problem" (not necessarily the input length).

Example: Knapsack

Objects $a_1, a_2, \ldots, a_n$ with weights $w_1, w_2, \ldots, w_n$ and values $s_1, s_2, \ldots, s_n$. B capacity of the knapsack. Assume $w_i \leq B$ for $1 \leq i \leq n$.

Aim (optimizing version): Select objects to pack into the knapsack such that it is not overloaded and the value is maximal.

Will see an algorithm with running time $O(nB)$. For large B, this is exponential in the length of the input (which is $O(\log w_1 + \cdots + \log w_n + \log s_1 + \cdots + \log s_n + \log B)$).

Pseudo-Polynomial Algorithms

- Make a 2-dimensional $(n+1) \times (B+1)$ table V.
- Let $A(i,w) \subseteq \{a_1, a_2, \ldots, a_i\}$ be a subset of the first i objects with maximal value and weight at most w.
- Let $V(i,w) = \sum_{a \in A(i,w)} s_i$ be the value of $A(i,w)$.
- In other words, $V(i,w)$ is the maximum value that can be composed with weight at most w using only objects up to i.
- The result is in cell $V(n,B)$.

Pseudo-Polynomial Algorithms

The following recurrence is used to fill the table in a so-called Dynamic Programming mode: solutions to restricted problems are used to solve less restricted problems.

- $V(0, w) = 0$, $V(i, 0) = 0$ (Initialization)
- $V(i + 1, w)$ is computed by distinguishing
  - If $w_{i+1} \leq w$ then object a_{i+1} might be used and $V(i + 1, w) = \max\{V(i, w) ; s_{i+1} + V(i, w - w_{i+1})\}$
  - If $w_{i+1} > w$ then object a_{i+1} cannot be used and $V(i + 1, w) = V(i, w)$

The time to fill the table is $O(nB)$.

Pseudo-Polynomial Algorithms

Example

$n = 4$, $W = \{2, 4, 3, 1\}$, $S = \{3, 4, 2, 1\}$, $B = 6$.

–	0	1	2	3	4	5	6
0	–	–	–	–	–	–	–
1	–	–	–	–	–	–	–
2	–	–	–	–	–	–	–
3	–	–	–	–	–	–	–
4	–	–	–	–	–	–	–

Pseudo-Polynomial Algorithms

Example

Initialization

$n = 4$, $W = \{2, 4, 3, 1\}$, $S = \{3, 4, 2, 1\}$, $B = 6$.

$-$	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	$-$	$-$	$-$	$-$	$-$	$-$
2	0	$-$	$-$	$-$	$-$	$-$	$-$
3	0	$-$	$-$	$-$	$-$	$-$	$-$
4	0	$-$	$-$	$-$	$-$	$-$	$-$

Pseudo-Polynomial Algorithms

Example

$n = 4$, $W = \{2, 4, 3, 1\}$, $S = \{3, 4, 2, 1\}$, $B = 6$.

–	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	–	–	–	–	–
2	0	–	–	–	–	–	–
3	0	–	–	–	–	–	–
4	0	–	–	–	–	–	–

For $V(1, 1)$: $w_1 = 2 > 1$ therefore $V(1, 1) = V(0, 1) = 0$

Pseudo-Polynomial Algorithms

Example

$n = 4$, $W = \{2, 4, 3, 1\}$, $S = \{3, 4, 2, 1\}$, $B = 6$.

–	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	3	–	–	–	–
2	0	–	–	–	–	–	–
3	0	–	–	–	–	–	–
4	0	–	–	–	–	–	–

For $V(1, 2)$: $w_1 = 2 \leq 2$ therefore
$V(1,2) = \max\{V(0,2)\,;\, s_1 + V(0, w - w_1)\} = \max\{V(0,2)\,;\, 3 + V(0,0)\} = \max\{0; 3+0\} = 3$

Pseudo-Polynomial Algorithms

Example

$n = 4$, $W = \{2, 4, 3, 1\}$, $S = \{3, 4, 2, 1\}$, $B = 6$.

–	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	3	3	3	3	3
2	0	0	3	3	4	4	7
3	0	0	3	3	4	5	7
4	0	1	3	4	4	5	7

The completely filled table; the maximum value attainable is 7.

Pseudo-Polynomial Algorithms

Example

The algorithm only computes the value.

In order to select the objects which attain this value, some more book keeping is needed: when computing $V(i+1, w)$ store whether a_{i+1} was used to achieve the maximum value.

A Second Pseudo-Polynomial Algorithm for Knapsack

The previous pseudo-polynomial algorithm is only efficient if B is small, i.e., the objects have small weights.

Now: an efficient algorithm for small object *values*.

- Let $S := s_1 + \cdots + s_n$.
- Make a 2-dimensional $(n+1) \times (S+1)$ table V.
- $V(i,s) := \min \left\{ \sum_{a_j \in A} w_j \mid A \subseteq \{a_1, \ldots, a_i\}, \sum_{a_j \in A} s_j = s \right\}$
- In other words, $V(i,s)$ is *minimum* weight that allows a total value of *exactly* s using only objects up to i.
- If this is impossible, $V(i,s) = \infty$.
- The result is the number of the rightmost colum j that contains an entry of at most B.

Filling the Table

- $V(0,0) = 0$, $V(0,s) = \infty$ for $s > 0$ (Initialization)
- $V(i+1,s)$ is computed by distinguishing
  - If $s_{i+1} \le s$ then object a_{i+1} might be used and $V(i+1,s) = \min\{V(i,s)\,;\,w_{i+1} + V(i, s - s_{i+1})\}$.
  - If $s_{i+1} > s$ then object a_{i+1} cannot be used and therefore $V(i+1,s) = V(i,s)$.

The time to fill the table is $O(nS)$.

Example

Consider the following problem instance.

$$n = 3 \quad W = \{6, 2, 2\} \quad S = \{4, 4, 2\} \quad B = 6$$

Then the final table would look like this and the output would be 6.

$-$	0	1	2	3	4	5	6	7	8	9	10
0	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	0	∞	∞	∞	6	∞	∞	∞	∞	∞	∞
2	0	∞	∞	∞	2	∞	∞	∞	8	∞	∞
3	0	∞	2	∞	2	∞	4	∞	8	∞	10

From Pseudo-Polynomial to Approximation Algorithm

Previous algorithm is efficient if S is small.

Idea: Divide all object values by some large number k, round down, apply algorithm, obtain solution.

Running time drops to $O(nS/k)$.

If k is not too big, the solution still has "good" quality for the unmodified instance.

Observe trade-off:
large $k \to$ small running time, but large rounding errors

Choosing the Approximation Quality

So far we have two pseudo-polynomial algorithms for Knapsack: runtime $O(nB)$ and $O(nS)$, respectively, which may be too slow.

Aim: fast algorithm reaching value s such that approximation ratio satisfies $\text{OPT}/s \leq 1+\epsilon$, where OPT is theoretical optimum.

Informally, solution is only by factor $1+\epsilon$ worse than optimum. ϵ is called error threshold.

Example: Choose $\epsilon = 0.01$, then solution only 1% off from the optimum. If $\text{OPT} = 100$, algorithm is *guaranteed* to compute solution of value $\geq 100/1.01 > 99$.

Algorithm with adjustable error threshold ϵ is often called an *approximation scheme.*

The Approximation Scheme

Given: objects $a_1, \ldots, a_n$ with weights $w_1, \ldots, w_n$ and values $s_1, \ldots, s_n$ and capacity B.

Step 1: Let

$$k := \frac{\epsilon \cdot \max\{s_1, \ldots, s_n\}}{(1+\epsilon)n}$$

and create a modified instance to the knapsack problem with the original weights $w_1, \ldots, w_n$ but modified values $\tilde{s}_1, \ldots, \tilde{s}_n$, where

$$\tilde{s}_i := \left\lfloor \frac{s_i}{k} \right\rfloor = \left\lfloor \frac{(1+\epsilon) \cdot n \cdot s_i}{\epsilon \cdot \max\{s_1, \ldots, s_n\}} \right\rfloor$$

for $1 \leq i \leq n$.

Moreover, capacity B is the same as in the original instance.

Step 2: Run second pseudo-polynomial algorithm (the one with the $n \times S$ table) on modified instance and use its solution for original problem. Why can we do that?

Analyzing the Running Time

Modified instance can be created in linear time. Running time of pseudo-polynomial algorithm is

$$O(n \cdot (\tilde{s}_1 + \cdots + \tilde{s}_n)) = O(n \cdot n \cdot \max\{\tilde{s}_1, \ldots, \tilde{s}_n\}),$$

which, by definition of the $\tilde{s}_i$, is

$$O\left(n^2 \cdot \max\left\{\left\lfloor \frac{(1+\epsilon) \cdot n \cdot s_1}{\epsilon \cdot \max\{s_1, \ldots, s_n\}} \right\rfloor, \ldots, \left\lfloor \frac{(1+\epsilon) \cdot n \cdot s_n}{\epsilon \cdot \max\{s_1, \ldots, s_n\}} \right\rfloor\right\}\right).$$

This the same as

$$O\left(n^2 \cdot \frac{(1+\epsilon) \cdot n}{\epsilon}\right) = O\left(\frac{n^3}{\epsilon} + n^3\right).$$

Altogether, running time is polynomial in the input length n and in $1/\epsilon$. If ϵ is a constant, then the running time is $O(n^3)$.