Algorithmics

Spring ’20 - Tutorial #5

Alexander Moch

May 11, 2020

Exercise 5.1

Task

Show that $\leq_p$ is a reflexive and transitive relation over $\mathcal{NP}$.

Hint: W.l.o.g., you can assume here that the polynomials which bound the worst-case running time of the corresponding transformations are monotonically increasing.

Exercise 5.1

To prove **Transitivity**, it must be shown that

$$((L_1 \leq_p L_2) \wedge (L_2 \leq_p L_3)) \Rightarrow L_1 \leq_p L_3.$$

Exercise 5.1

The preconditions state that there are functions

$$f_1 : \Sigma_1^* \to \Sigma_2^* \text{ and}$$
$$f_2 : \Sigma_2^* \to \Sigma_3^*,$$

that can be computed in polynomial worst-case time p_1, resp. p_2, and for which it holds:

$$\forall x \in \Sigma_1^* : x \in L_1 \Leftrightarrow f_1(x) \in L_2 \text{ and}$$
$$\forall y \in \Sigma_2^* : y \in L_2 \Leftrightarrow f_2(y) \in L_3.$$

Exercise 5.1

- W.l.o.g. let p_2 be monotonically increasing.

Exercise 5.1

- W.l.o.g. let p_2 be monotonically increasing.
- We define $f_3 : \Sigma_1^* \to \Sigma_3^*$ to be $f_3 := f_2 \circ f_1$.

Exercise 5.1

- W.l.o.g. let p_2 be monotonically increasing.
- We define $f_3 : \Sigma_1^* \to \Sigma_3^*$ to be $f_3 := f_2 \circ f_1$.
- Since $|f_1(x)| \leq p_1(|x|)$, f_3 is computable in polynomial worst case time $p_1 + p_2 \circ p_1$ and it holds:

$$\forall x \in \Sigma_1^* : x \in L_1 \Leftrightarrow f_1(x) \in L_2 \Leftrightarrow f_2 \circ f_1(x) \in L_3 \Leftrightarrow f_3(x) \in L_3.$$

Exercise 5.1

To prove **Reflexivity**, we choose

$$f := id.$$

Exercise 5.2

Exercise 5.2

Show that PARTITION $\in \mathcal{NPC}$ by showing that KP* $\leq_p$ PARTITION.

Exercise 5.2

Show that PARTITION $\in \mathcal{NPC}$ by showing that KP* $\leq_p$ PARTITION.

(KP* $\in \mathcal{NPC}$ has already been shown in detail in the lecture.)

Exercise 5.2

PARTITION $\in \mathcal{NP}$ is trivial.[1]

[1] Easy exercise: what does the corresponding proof scheme look like?

Exercise 5.2

PARTITION $\in \mathcal{NP}$ is trivial.[1]

In the lecture, we proved that the following special knapsack problem KP* is $\mathcal{NP}$-complete:

[1] Easy exercise: what does the corresponding proof scheme look like?

Exercise 5.2

PARTITION $\in \mathcal{NP}$ is trivial.[1]

In the lecture, we proved that the following special knapsack problem KP* is $\mathcal{NP}$-complete:

Given $\vec{w} := (w_1, \ldots, w_n) \in \mathbb{N}^n$ and $W \in \mathbb{N}$:

Decide whether there exists an $I \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in I} w_i = W$.

[1] Easy exercise: what does the corresponding proof scheme look like?

Exercise 5.2

It is now sufficient to show that KP* $\leq_p$ PARTITION holds by providing a corresponding polynomial reduction *f*.

Exercise 5.2

It is now sufficient to show that KP* $\leq_p$ PARTITION holds by providing a corresponding polynomial reduction f.

Let $(\vec{w}, W)$, where $\vec{w} := (w_1, \ldots, w_n) \in \mathbb{N}^n$ and $W \in \mathbb{N}$, be an input instance for KP*.

Based on it, we compute the input instance for PARTITION in polynomial time:

$f((\vec{w}, W)) := (w_1, \ldots, w_n, S - W + 1, W + 1)$, where $S := \sum_{i=1}^{n} w_i$.

Solution

$(\vec{w}, W) \in \text{KP}^* \Rightarrow f((\vec{w}, W)) \in \text{PARTITION}$:

If I is a solution for the problem KP*, then $I \cup \{n+1\}$ constitutes a solution for PARTITION because

$$\begin{aligned}\left(\sum_{i \in I} w_i\right) + (S - W + 1) &= W + S - W + 1 \\ &= S - W + W + 1 \\ &= \left(\sum_{i=1}^{n} w_i\right) - \left(\sum_{i \in I} w_i\right) + W + 1 \\ &= \left(\sum_{i \notin I} w_i\right) + (W + 1)\,.\end{aligned}$$

Exercise 5.2

$f\big((\vec{w}, W)\big) \in \text{PARTITION} \Rightarrow (\vec{w}, W) \in \text{RP}^*$:

- The sum of all numbers in the input for PARTITION equals $2S + 2$.

Exercise 5.2

$f\big((\vec{w}, W)\big) \in \text{PARTITION} \Rightarrow (\vec{w}, W) \in \text{RP}^*$:

- The sum of all numbers in the input for PARTITION equals $2S + 2$.
- A solution for PARTITION has to satisfy the condition that each of the two parts amounts to $S + 1$.

Exercise 5.2

$f\big((\vec{w}, W)\big) \in \text{PARTITION} \Rightarrow (\vec{w}, W) \in \text{RP}^*$:

- The sum of all numbers in the input for PARTITION equals $2S + 2$.
- A solution for PARTITION has to satisfy the condition that each of the two parts amounts to $S + 1$.
- For this reason, the numbers $S - W + 1$ and $W + 1$ have to be in different parts.

Exercise 5.2

$f\big((\vec{w}, W)\big) \in \text{PARTITION} \Rightarrow (\vec{w}, W) \in \text{RP}^*$:

- The sum of all numbers in the input for PARTITION equals $2S + 2$.
- A solution for PARTITION has to satisfy the condition that each of the two parts amounts to $S + 1$.
- For this reason, the numbers $S - W + 1$ and $W + 1$ have to be in different parts.
- The numbers which complement $S - W + 1$ to $S + 1$ have the sum W and represent a solution for the input of KP*.

Exercise 5.3

Task

Show: The optimization versions of TSP and KP are $\mathcal{NP}$-easy.

Solution - Self-Reduction MinTSP $\leq_T$ TSP

Polynomial-time TSP-algorithm for MinTSP (1):

Solution - Self-Reduction MinTSP $\leq_T$ TSP

Polynomial-time TSP-algorithm for MinTSP (1):

- We use the decision version of TSP as an oracle from $\mathcal{NP}$.

Solution - Self-Reduction MinTSP $\leq_T$ TSP

Polynomial-time TSP-algorithm for MinTSP (1):

- We use the decision version of TSP as an oracle from $\mathcal{NP}$.
- First of all, we compute the costs c_{opt} of an optimal round trip with the help of the oracle.

Solution - Self-Reduction MinTSP $\leq_T$ TSP

Polynomial-time TSP-algorithm for MinTSP (1):

- We use the decision version of TSP as an oracle from $\mathcal{NP}$.
- First of all, we compute the costs c_{opt} of an optimal round trip with the help of the oracle.
- Let n be the number of vertices of the input instance and c_{max} the weight of the most expensive edge.

Solution - Self-Reduction MinTSP $\leq_T$ TSP

Polynomial-time TSP-algorithm for MinTSP (1):

- We use the decision version of TSP as an oracle from $\mathcal{NP}$.
- First of all, we compute the costs c_{opt} of an optimal round trip with the help of the oracle.
- Let n be the number of vertices of the input instance and c_{max} the weight of the most expensive edge.
- Then $n \cdot c_{max}$ is an upper bound for the costs of every round trip.

Solution - Self-Reduction MinTSP $\leq_T$ TSP

Polynomial-time TSP-algorithm for MinTSP (1):

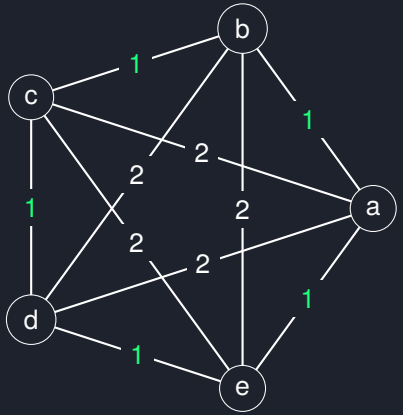
- We use the decision version of TSP as an oracle from $\mathcal{NP}$.
- First of all, we compute the costs c_{opt} of an optimal round trip with the help of the oracle.
- Let n be the number of vertices of the input instance and c_{max} the weight of the most expensive edge.
- Then $n \cdot c_{max}$ is an upper bound for the costs of every round trip.
- Using binary search, $r := \lceil \log(n \cdot c_{max} + 1) \rceil$ oracle queries are sufficient for computing c_{opt}.

Solution - Self-Reduction MinTSP $\leq_T$ TSP

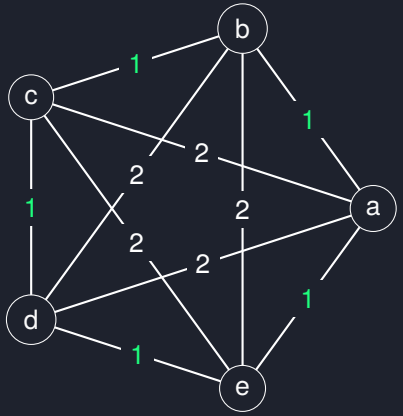
Polynomial-time TSP-algorithm for MinTSP (1):

- We use the decision version of TSP as an oracle from $\mathcal{NP}$.
- First of all, we compute the costs c_{opt} of an optimal round trip with the help of the oracle.
- Let n be the number of vertices of the input instance and c_{max} the weight of the most expensive edge.
- Then $n \cdot c_{max}$ is an upper bound for the costs of every round trip.
- Using binary search, $r := \lceil \log(n \cdot c_{max} + 1) \rceil$ oracle queries are sufficient for computing c_{opt}.
- Note that r is polynomial in the length of the input.

Example: MinTSP $\leq_T$ TSP

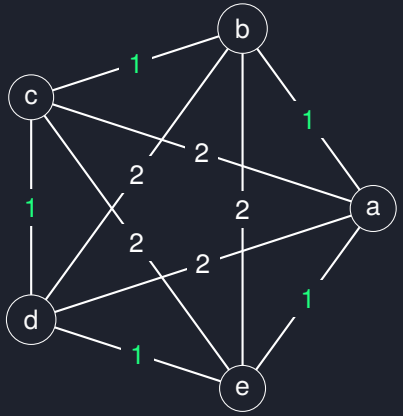


Example: MinTSP $\leq_T$ TSP



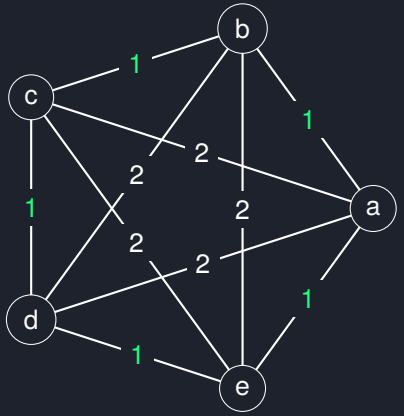
Is there a roundtrip of cost ≤ 10? $\overset{\mathcal{O}}{\to}$ Yes

Example: MinTSP $\leq_T$ TSP



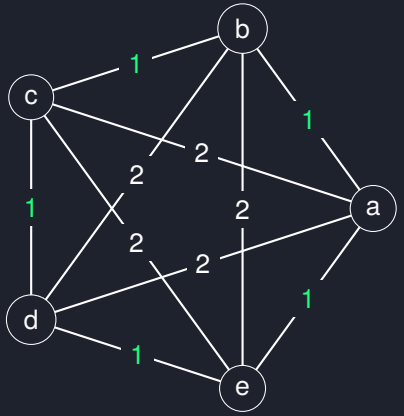
Is there a roundtrip of cost ≤ 10? $\xrightarrow{\mathcal{O}}$ Yes
Is there a roundtrip of cost ≤ 5? $\xrightarrow{\mathcal{O}}$ Yes

Example: MinTSP $\leq_T$ TSP



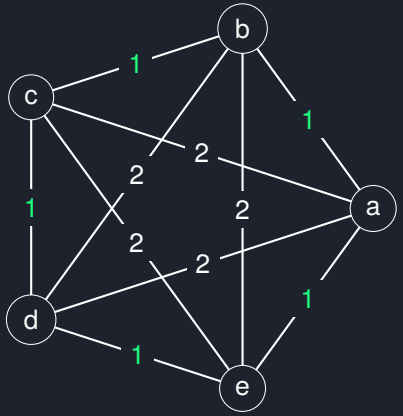
Is there a roundtrip of cost ≤ 10? $\overset{\mathcal{O}}{\rightarrow}$ Yes
Is there a roundtrip of cost ≤ 5? $\overset{\mathcal{O}}{\rightarrow}$ Yes
Is there a roundtrip of cost ≤ 3? $\overset{\mathcal{O}}{\rightarrow}$ No

Example: MinTSP $\leq_T$ TSP



Is there a roundtrip of cost ≤ 10? $\overset{\mathcal{O}}{\rightarrow}$ Yes
Is there a roundtrip of cost ≤ 5? $\overset{\mathcal{O}}{\rightarrow}$ Yes
Is there a roundtrip of cost ≤ 3? $\overset{\mathcal{O}}{\rightarrow}$ No
Is there a roundtrip of cost ≤ 4? $\overset{\mathcal{O}}{\rightarrow}$ No

Example: MinTSP $\leq_T$ TSP



Is there a roundtrip of cost ≤ 10? $\overset{\mathcal{O}}{\to}$ Yes
Is there a roundtrip of cost ≤ 5? $\overset{\mathcal{O}}{\to}$ Yes
Is there a roundtrip of cost ≤ 3? $\overset{\mathcal{O}}{\to}$ No
Is there a roundtrip of cost ≤ 4? $\overset{\mathcal{O}}{\to}$ No
$\Longrightarrow$ The optimal roundtrip has a cost of 5.

Solution - Self-Reduction MinTSP $\leq_T$ TSP

Polynomial-time TSP-algorithm for MinTSP (2):

Solution - Self-Reduction MinTSP $\leq_T$ TSP

Polynomial-time TSP-algorithm for MinTSP (2):

- Subsequently, at most $n(n-1)$ additional oracle queries are sufficient for computing an optimal round trip.

Solution - Self-Reduction MinTSP $\leq_T$ TSP

Polynomial-time TSP-algorithm for MinTSP (2):

- Subsequently, at most $n(n-1)$ additional oracle queries are sufficient for computing an optimal round trip.
- More precisely, we temporarily “remove” all edges one by one, i.e., we replace their actual weight by $c_{opt} + 1$.

Solution - Self-Reduction MinTSP $\leq_T$ TSP

Polynomial-time TSP-algorithm for MinTSP (2):

- Subsequently, at most $n(n-1)$ additional oracle queries are sufficient for computing an optimal round trip.
- More precisely, we temporarily “remove” all edges one by one, i.e., we replace their actual weight by $c_{opt} + 1$.
- For each of the edges (i.e., in each step), we use the oracle to check whether there still exists a round trip with (optimal) costs c_{opt} after setting the edge’s weight to $c_{opt} + 1$.

Solution - Self-Reduction MinTSP $\leq_T$ TSP

Polynomial-time TSP-algorithm for MinTSP (2):

- Subsequently, at most $n(n-1)$ additional oracle queries are sufficient for computing an optimal round trip.
- More precisely, we temporarily "remove" all edges one by one, i.e., we replace their actual weight by $c_{opt} + 1$.
- For each of the edges (i.e., in each step), we use the oracle to check whether there still exists a round trip with (optimal) costs c_{opt} after setting the edge's weight to $c_{opt} + 1$.
- If this is not the case, then the respective edge is necessary for an optimal round trip and we set the weight of the edge back to its initial value.

Solution - Self-Reduction MinTSP $\leq_T$ TSP

Polynomial-time TSP-algorithm for MinTSP (2):

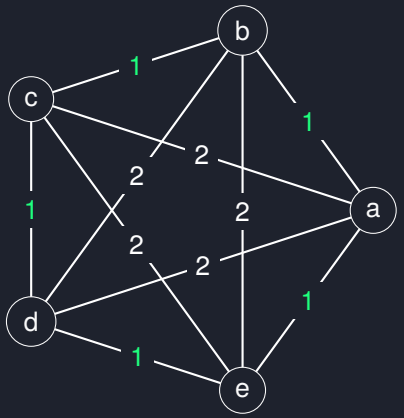
- Subsequently, at most $n(n-1)$ additional oracle queries are sufficient for computing an optimal round trip.
- More precisely, we temporarily "remove" all edges one by one, i.e., we replace their actual weight by $c_{opt} + 1$.
- For each of the edges (i.e., in each step), we use the oracle to check whether there still exists a round trip with (optimal) costs c_{opt} after setting the edge's weight to $c_{opt} + 1$.
- If this is not the case, then the respective edge is necessary for an optimal round trip and we set the weight of the edge back to its initial value.
- Otherwise, we keep the (new) weight $c_{opt} + 1$ for this edge.

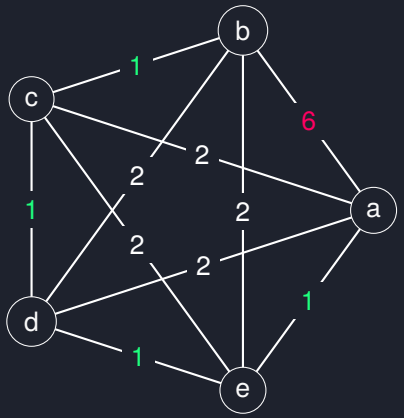
Solution - Self-Reduction MinTSP $\leq_T$ TSP

Polynomial-time TSP-algorithm for MinTSP (2):

- Subsequently, at most $n(n-1)$ additional oracle queries are sufficient for computing an optimal round trip.
- More precisely, we temporarily "remove" all edges one by one, i.e., we replace their actual weight by $c_{opt} + 1$.
- For each of the edges (i.e., in each step), we use the oracle to check whether there still exists a round trip with (optimal) costs c_{opt} after setting the edge's weight to $c_{opt} + 1$.
- If this is not the case, then the respective edge is necessary for an optimal round trip and we set the weight of the edge back to its initial value.
- Otherwise, we keep the (new) weight $c_{opt} + 1$ for this edge.
- After treating all edges in this manner, those edges whose weight was not set to $c_{opt} + 1$ constitute an optimal round trip.

Example: MinTSP $\leq_T$ TSP

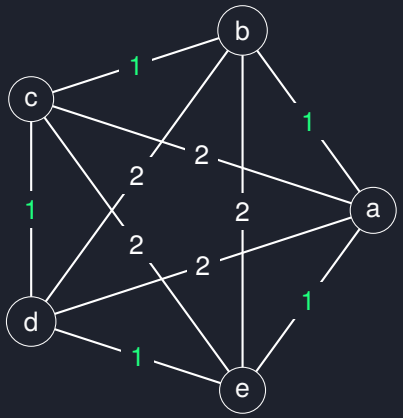


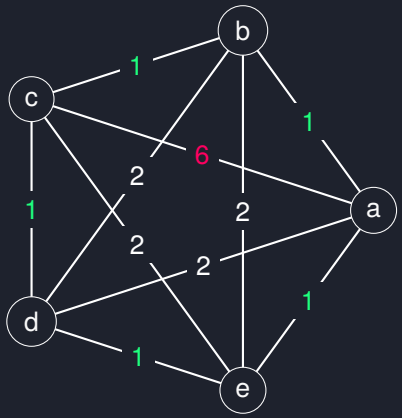


Is there a roundtrip of cost 5? $\rightarrow$ No.

$\Longrightarrow$ Edge (a, b) is part of the optimal roundtrip.

Example: MinTSP $\leq_T$ TSP

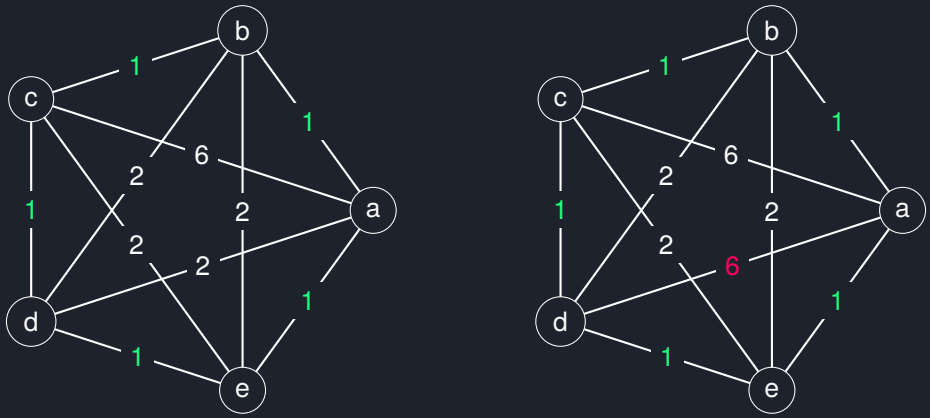




Is there a roundtrip of cost 5? $\rightarrow$ Yes.

$\Longrightarrow$ Edge (a, c) is not part of the optimal roundtrip.

Example: MinTSP $\leq_T$ TSP



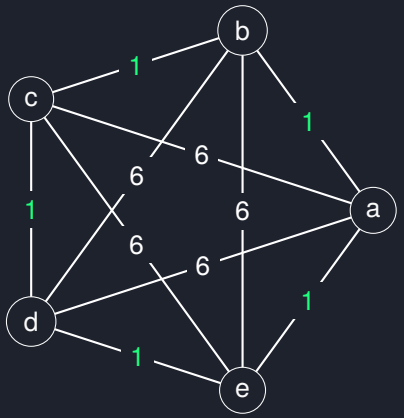
Is there a roundtrip of cost 5? $\rightarrow$ Yes.

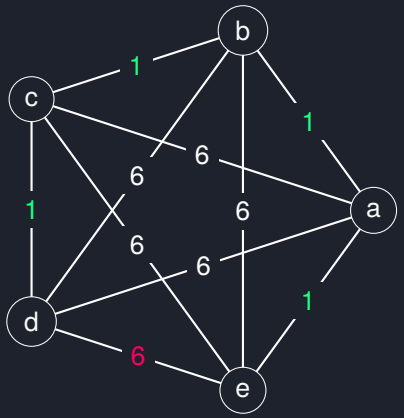
$\Longrightarrow$ Edge (a, d) is not part of the optimal roundtrip.

Example: MinTSP $\leq_T$ TSP

...

Example: MinTSP $\leq_T$ TSP

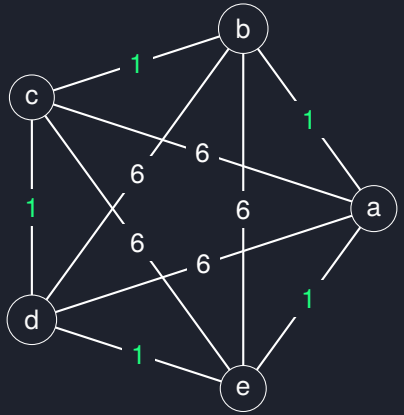




Is there a roundtrip of cost 5? $\rightarrow$ No.

$\Longrightarrow$ Edge (d, e) is part of the optimal roundtrip.

Example: MinTSP $\leq_T$ TSP



All Edges with weight $\neq 6$ are part of the optimal roundtrip.

Solution - Self-Reduction MaxKP $\leq_T$ KP

Reminder:

KP:

- **Input:** $X = (n, \vec{w}, \vec{c}, W, C)$, where $n, W, C \in \mathbb{N}$, $\vec{w} = (w_1, \ldots, w_n) \in \mathbb{N}^n$, $\vec{c} = (c_1, \ldots, c_n) \in \mathbb{N}^n$.
- **Accept** X iff $\exists I \subseteq \{1, \ldots, n\}$ with $w(I) := \sum_{i \in I} w_i \leq W$ and $c(I) := \sum_{i \in I} c_i \geq C$.

MaxKP:

- **Input:** $X = (n, \vec{w}, \vec{c}, W)$.
- **Output:** $I^* \subseteq \{1, \ldots, n\}$ with $w(I^*) \leq W$ and $c(I^*) = opt(X) := \max\{c(I) \mid I \subseteq \{1, \ldots, n\},\ w(I) \leq W\}$.

Solution - Self-Reduction MaxKP $\leq_T$ KP

Polynomial-time KP-algorithm for MaxKP:

Solution - Self-Reduction MaxKP $\leq_T$ KP

Polynomial-time KP-algorithm for MaxKP:

Input: $X_{\text{MaxKP}} = (n, \vec{w}, \vec{c}, W)$

Solution - Self-Reduction MaxKP $\leq_T$ KP

Polynomial-time KP-algorithm for MaxKP:

Input: $X_{\text{MaxKP}} = (n, \vec{w}, \vec{c}, W)$

Step 1: Compute $C^* := opt(X_{\text{MaxKP}})$ using oracle-based binary search in the interval $0, \ldots, \sum_{i=1}^{n} c_i$.

Solution - Self-Reduction MaxKP $\leq_T$ KP

Polynomial-time KP-algorithm for MaxKP:

Input: $X_{\text{MaxKP}} = (n, \vec{w}, \vec{c}, W)$

Step 1: Compute $C^* := opt(X_{\text{MaxKP}})$ using oracle-based binary search in the interval $0, \ldots, \sum_{i=1}^{n} c_i$.

Step 2:

$X'_{\text{KP}} := (n, \vec{w}, \vec{c}, W, C^*)$
for all $i = 1, \ldots, n$ **do**
  $X''_{\text{KP}} := X'_{\text{KP}} -$ "Object i of X_{MaxKP}" (W and C^* do not change)
  if $X''_{\text{KP}} \in$ KP **then**
    $X'_{\text{KP}} := X''_{\text{KP}}$
output "Index set $I^* \subseteq \{1, \ldots, n\}$ of all those objects of X_{MaxKP} finally remaining in X'_{KP}"

Solution - Self-Reduction MaxKP $\leq_T$ KP

Instead of actually removing objects, we could have also used the following approach (similar to our “MinTSP $\leq_T$ TSP” reduction).

Solution - Self-Reduction MaxKP $\leq_T$ KP

Instead of actually removing objects, we could have also used the following approach (similar to our "MinTSP $\leq_T$ TSP" reduction).

Polynomial-time KP-algorithm for MaxKP:

Solution - Self-Reduction MaxKP $\leq_T$ KP

Instead of actually removing objects, we could have also used the following approach (similar to our "MinTSP $\leq_T$ TSP" reduction).

Polynomial-time KP-algorithm for MaxKP:

Input: $X_{\text{MaxKP}} = (n, \vec{w}, \vec{c}, W)$

Solution - Self-Reduction MaxKP $\leq_T$ KP

Instead of actually removing objects, we could have also used the following approach (similar to our "MinTSP $\leq_T$ TSP" reduction).

Polynomial-time KP-algorithm for MaxKP:

Input: $X_{\text{MaxKP}} = (n, \vec{w}, \vec{c}, W)$

Step 1: Compute $C^* := opt(X_{\text{MaxKP}})$ using oracle-based binary search in the interval $0, \ldots, \sum_{i=1}^{n} c_i$.

Solution - Self-Reduction MaxKP $\leq_T$ KP

Instead of actually removing objects, we could have also used the following approach (similar to our "MinTSP $\leq_T$ TSP" reduction).

Polynomial-time KP-algorithm for MaxKP:

Input: $X_{\text{MaxKP}} = (n, \vec{w}, \vec{c}, W)$

Step 1: Compute $C^* := opt(X_{\text{MaxKP}})$ using oracle-based binary search in the interval $0, \ldots, \sum_{i=1}^{n} c_i$.

Step 2:
$(w'_1, \ldots, w'_n) := \vec{w}$
for all $i = 1, \ldots, n$ **do**
  if $(n, (w'_1, \ldots, w'_{i-1}, W+1, w'_{i+1}, \ldots, w'_n), \vec{c}, W, C^*) \in \text{KP}$ **then**
    $w'_i := W + 1$
output $\{i \mid w'_i \neq W + 1\}$

Exercise 5.4

Task

For the BIN PACKING PROBLEM (BPP), each instance consists of

- a set of objects $A = \{a_1, \ldots, a_n\}$,
- a function $s : A \to \mathbb{N}$, which assigns a size/weight to each of the objects,
- and a container size $B \in \mathbb{N}$.

The question asked in the optimization version is, how these objects can be packed into as few containers of capacity B as possible.

More precisely, we look for a partition $A_1 \dot\cup \cdots \dot\cup A_m = A$ such that $\max\{\sum_{a_i \in A_j} s(a_i) \mid 1 \leq j \leq m\} \leq B$ holds and m is as small as possible.

In the decision version, we are given an additional number $K \in \{1, \ldots, n\}$ and ask whether K containers are sufficient for storing all objects.

The Bin Packing Problem

You are given a set of objects $\underset{w}{x}$ with weight w. For example:

$$\underset{6}{a}, \underset{5}{b}, \underset{13}{c}, \underset{5}{d}, \underset{8}{e}, \underset{5}{f}$$

Decision Problem: Is it possible to pack the objects into k bins of size s?

Optimization Problem: What is the minimum amount of bins of size s to pack the objects?

If the bin size is $s = 15$, the object can be packed into $k = 3$ bins:

$$\{\underset{6}{a}, \underset{8}{e}\} \{\underset{5}{b}, \underset{5}{d}, \underset{5}{f}\} \{\underset{13}{c}\}$$

Task

Show:

a) The decision version of BPP is $\mathcal{NP}$-complete.

b) The optimization version of BPP is $\mathcal{NP}$-equivalent.

Solution (a)

Obviously, the decision version of BPP, which we will denote DecBPP for the sake of clarity, is in $\mathcal{NP}$ (easy exercise: what does the corresponding proof scheme look like?).

Solution (a)

Obviously, the decision version of BPP, which we will denote DecBPP for the sake of clarity, is in $\mathcal{NP}$ (easy exercise: what does the corresponding proof scheme look like?).

DecBPP is now $\mathcal{NP}$-complete because the $\mathcal{NP}$-complete problem PARTITION is a special case of it (hence, PARTITION $\leq_p$ DecBPP).

Solution (a)

Obviously, the decision version of BPP, which we will denote DecBPP for the sake of clarity, is in $\mathcal{NP}$ (easy exercise: what does the corresponding proof scheme look like?).

DecBPP is now $\mathcal{NP}$-complete because the $\mathcal{NP}$-complete problem PARTITION is a special case of it (hence, PARTITION $\leq_p$ DecBPP).

Concretely, an input instance $(p_1, \ldots, p_n) \in \mathbb{N}^n$ for PARTITION can be transformed into the input instance (A, s, B, K) with

- $A := \{a_1, \ldots, a_n\}$,
- $s(a_i) := p_i$,
- $B := \frac{1}{2} \sum_{i=1}^{n} s(a_i)$,
- $K := 2$

for DecBPP.

Solution (b)

- It is easy to see that the optimization version, which we will denote MinBPP for the sake of clarity, is $\mathcal{NP}$-hard because the decision version DecBPP is $\mathcal{NP}$-complete.

Solution (b)

- It is easy to see that the optimization version, which we will denote MinBPP for the sake of clarity, is $\mathcal{NP}$-hard because the decision version DecBPP is $\mathcal{NP}$-complete.
- We now show that MinBPP is $\mathcal{NP}$-easy by providing a polynomial Turing reduction from MinBPP to DecBPP.

Solution (b)

- It is easy to see that the optimization version, which we will denote MinBPP for the sake of clarity, is $\mathcal{NP}$-hard because the decision version DecBPP is $\mathcal{NP}$-complete.
- We now show that MinBPP is $\mathcal{NP}$-easy by providing a polynomial Turing reduction from MinBPP to DecBPP.
- We suppose that the regarded input instance possesses a solution.

Solution (b)

- It is easy to see that the optimization version, which we will denote MinBPP for the sake of clarity, is $\mathcal{NP}$-hard because the decision version DecBPP is $\mathcal{NP}$-complete.
- We now show that MinBPP is $\mathcal{NP}$-easy by providing a polynomial Turing reduction from MinBPP to DecBPP.
- We suppose that the regarded input instance possesses a solution.
- Oracle queries (i.e., asking whether for a given K, the objects in A fit into K containers of size B) can be employed to compute the minimal number of needed containers in polynomial time (through linear or binary search in the interval $1, \ldots, n$; as, in the worst case, each of the n objects in A would need a container of its own). (*Step 1* of our algorithm)

Solution (b) - Self-Reduction MinBPP $\leq_T$ DecBPP

Polynomial-time DecBPP-algorithm for MinBPP:

Solution (b) - Self-Reduction MinBPP $\leq_T$ DecBPP

Polynomial-time DecBPP-algorithm for MinBPP:

Input: $I = (A, s, B)$

Solution (b) - Self-Reduction MinBPP $\leq_T$ DecBPP

Polynomial-time DecBPP-algorithm for MinBPP:

Input: $I = (A, s, B)$

Step 1: Compute $K^* := opt(I)$ (where $opt(I)$ denotes the minimal number of needed containers for the MinBPP input instance I) using oracle-based linear or binary search in the interval $1, \ldots, n$.

Solution (b) - Self-Reduction MinBPP $\leq_T$ DecBPP

Polynomial-time DecBPP-algorithm for MinBPP:

Input: $I = (A, s, B)$

Step 1: Compute $K^* := opt(I)$ (where $opt(I)$ denotes the minimal number of needed containers for the MinBPP input instance I) using oracle-based linear or binary search in the interval $1, \ldots, n$.

Step 2: See next slide.

MinBPP Oracle Algorithm

Step 2:

$A' := A$
$P := \{\}$
$j := 0$
repeat until $A' = \emptyset$
  $j := j + 1$
  $x := GetArbitraryElementFromSet(A')$ (x is an arbitrary element from the set A')
  $A' := A' \setminus \{x\}$
  $T := \{x\}$
  $t := s(x)$
  $\hat{A} := A'$
  for each $y \in \hat{A}$ **do**
    $z := CreateNewObjectNotInSet(A)$ (z is a new, temporary object that fulfills $z \notin A$)
    $\tilde{A} := \left(A' \setminus \{y\}\right) \cup \{z\}$
    $\tilde{s}(e) := \begin{cases} s(e), & \text{if } e \in \tilde{A} \setminus \{z\} \\ t + s(y), & \text{if } e = z \end{cases}$ (the function $\tilde{s}$ maps $\tilde{A}$ to $\mathbb{N}$)
    $\tilde{K} := K^* - (j - 1)$
    if $(\tilde{A}, \tilde{s}, B, \tilde{K}) \in$ DecBPP **then**
      $A' := A' \setminus \{y\}$
      $T := T \cup \{y\}$
      $t := t + s(y)$
  $P := P \cup \{T\}$ (note that P is a set of pairwise disjoint sets of objects)
output P (P now represents the optimal partition w.r.t. the MinBPP input instance I and consists of j sets)

The BPP Oracle Algorithm - First Iteration

a	b	c	d	e	f	$\mathcal{O}$
6	5	13	5	8	5	3 bins

The BPP Oracle Algorithm - First Iteration

a 6	b 5	c 13	d 5	e 8	f 5	$\mathcal{O}$ 3 bins
$[a, b]$ 11	~~b 5~~	c 13	d 5	e 8	f 5	No

The BPP Oracle Algorithm - First Iteration

a 6	b 5	c 13	d 5	e 8	f 5	$\mathcal{O}$ 3 bins
$[a, b]$ 11	~~b 5~~	c 13	d 5	e 8	f 5	No
$[a, c]$ 19	b 5	~~c 13~~	d 5	e 8	f 5	No

The BPP Oracle Algorithm - First Iteration

a 6	b 5	c 13	d 5	e 8	f 5	$\mathcal{O}$ 3 bins
$[a, b]$ 11	~~b 5~~	c 13	d 5	e 8	f 5	No
$[a, c]$ 19	b 5	~~c 13~~	d 5	e 8	f 5	No
$[a, d]$ 11	b 5	c 13	~~d 5~~	e 8	f 5	No

The BPP Oracle Algorithm - First Iteration

a 6	b 5	c 13	d 5	e 8	f 5	$\mathcal{O}$ 3 bins
$[a, b]$ 11	~~b 5~~	c 13	d 5	e 8	f 5	No
$[a, c]$ 19	b 5	~~c 13~~	d 5	e 8	f 5	No
$[a, d]$ 11	b 5	c 13	~~d 5~~	e 8	f 5	No
$[a, e]$ 14	b 5	c 13	d 5	~~e 8~~	f 5	Yes

The BPP Oracle Algorithm - First Iteration

a 6	b 5	c 13	d 5	e 8	f 5	$\mathcal{O}$ 3 bins
$[a, b]$ 11	~~b 5~~	c 13	d 5	e 8	f 5	No
$[a, c]$ 19	b 5	~~c 13~~	d 5	e 8	f 5	No
$[a, d]$ 11	b 5	c 13	~~d 5~~	e 8	f 5	No
$[a, e]$ 14	b 5	c 13	d 5	~~e 8~~	f 5	Yes
$[a, e, f]$ 19	b 5	c 13	d 5	~~e 8~~	~~f 5~~	No

The BPP Oracle Algorithm - Second Iteration

a	b	c	d	e	f	$\mathcal{O}$
6	5	13	5	8	5	2 bins

The BPP Oracle Algorithm - Second Iteration

a 6	b 5	c 13	d 5	e 8	f 5	$\mathcal{O}$ 2 bins
~~a 6~~	$[b, c]$ 18	~~c 13~~	d 5	~~e 8~~	f 5	No

The BPP Oracle Algorithm - Second Iteration

a 6	b 5	c 13	d 5	e 8	f 5	$\mathcal{O}$ 2 bins
~~a 6~~	$[b, c]$ 18	~~c 13~~	d 5	~~e 8~~	f 5	No
~~a 6~~	$[b, d]$ 10	c 13	~~d 5~~	~~e 8~~	f 5	Yes

The BPP Oracle Algorithm - Second Iteration

a 6	b 5	c 13	d 5	e 8	f 5	$\mathcal{O}$ 2 bins
~~a~~ ~~6~~	$[b, c]$ 18	~~c~~ ~~13~~	d 5	~~e~~ ~~8~~	f 5	No
~~a~~ ~~6~~	$[b, d]$ 10	c 13	~~d~~ ~~5~~	~~e~~ ~~8~~	f 5	Yes
~~a~~ ~~6~~	$[b, d, f]$ 15	c 13	~~d~~ ~~5~~	~~e~~ ~~8~~	~~f~~ ~~5~~	Yes

The BPP Oracle Algorithm - Second Iteration

a 6	b 5	c 13	d 5	e 8	f 5	$\mathcal{O}$ 2 bins
~~a 6~~	$[b, c]$ 18	~~c 13~~	d 5	~~e 8~~	f 5	No
~~a 6~~	$[b, d]$ 10	c 13	~~d 5~~	~~e 8~~	f 5	Yes
~~a 6~~	$[b, d, f]$ 15	c 13	~~d 5~~	~~e 8~~	~~f 5~~	Yes

$\{a, e\}$, $\{b, d, f\}$, $\{c\}$ are the three bins.

Exercise 5.5

Task

Which of the following statements are true? Why?

a) $L_1 \leq_p L_2 \Rightarrow \overline{L}_1 \leq_p \overline{L}_2$,

b) $(L_1 \leq_p L_2 \ \wedge \ L_2 \in \mathcal{P}) \Rightarrow L_1 \in \mathcal{P}$,

c) $(L_1 \leq_p L_2 \ \wedge \ L_2 \in \mathcal{NP}) \Rightarrow L_1 \in \mathcal{NP}$,

d) $(L \in \mathcal{NP} \ \wedge \ L \notin \mathcal{P}) \Rightarrow \mathcal{P} \cap \mathcal{NPC} = \emptyset$.

Solution

Which of the following statements are true? Why?

a) $L_1 \leq_p L_2 \Rightarrow \overline{L}_1 \leq_p \overline{L}_2$,
b) $(L_1 \leq_p L_2 \ \wedge \ L_2 \in \mathcal{P}) \Rightarrow L_1 \in \mathcal{P}$,
c) $(L_1 \leq_p L_2 \ \wedge \ L_2 \in \mathcal{NP}) \Rightarrow L_1 \in \mathcal{NP}$,
d) $(L \in \mathcal{NP} \ \wedge \ L \notin \mathcal{P}) \Rightarrow \mathcal{P} \cap \mathcal{NPC} = \emptyset$.

---

a) True: As $(x \in L_1 \Leftrightarrow f(x) \in L_2) \Longleftrightarrow (x \notin L_1 \Leftrightarrow f(x) \notin L_2) = (x \in \overline{L}_1 \Leftrightarrow f(x) \in \overline{L}_2)$.
b) True: First reduce L_1 to L_2 in polynomial time, then decide L_2 in polynomial time.
c) True: See Exercise 4.2.
d) True: $(L \in \mathcal{NP} \ \wedge \ L \notin \mathcal{P})$ implies $\mathcal{P} \neq \mathcal{NP}$. In the lecture, we proved that

$$\mathcal{P} \neq \mathcal{NP} \Rightarrow \mathcal{P} \cap \mathcal{NPC} = \emptyset.$$


That's all Folks!