Journal of Computational and Applied Mathematics 235 (2011) 4851–4865

Contents lists available at ScienceDirect

Journal of Computational and Applied Mathematics

journal homepage: www.elsevier.com/locate/cam

Survey of polynomial transformations between NP-complete problems

Jorge A. Ruiz-Vanoye [a,*], Joaquín Pérez-Ortega [b], Rodolfo A. Pazos R. [c], Ocotlán Díaz-Parra [d], Juan Frausto-Solís [e], Hector J. Fraire Huacuja [c], Laura Cruz-Reyes [c], José A. Martínez F. [c]

[a] Universidad Popular Autónoma del Estado de Puebla, 21 sur 1103 Barrio de Santiago, Puebla, Puebla, Mexico

[b] Centro Nacional de Investigación y Desarrollo Tecnológico, Interior internado Palmira s/n, Col. Palmira, Cuernavaca, Morelos, Mexico

[c] Instituto Tecnológico de Ciudad Madero, Av. 1o. de Mayo, Col. Los Mangos, Ciudad Madero, Tamaulipas, Mexico

[d] Universidad Autónoma del Estado de Morelos, Av. Universidad 1001, Col. Chamilpa, Cuernavaca, Morelos, Mexico

[e] Tecnológico de Monterrey Campus Cuernavaca, Autopista del Sol km 104, Colonia Real del Puente, Xochitepec, Morelos, Mexico

ARTICLE INFO

Keywords:
Polynomial transformations
NP-complete
Survey

ABSTRACT

This paper aims at being a guide to understand polynomial transformations and polynomial reductions between NP-complete problems by presenting the methodologies for polynomial reductions/transformations and the differences between reductions and transformations. To this end the article shows examples of polynomial reductions/transformations and the restrictions to reduce/transform between NP-complete problems. Finally, this paper includes a digraph with the historical reductions/transformations between instances of NP-complete problems and introduces the term family of polynomial transformations.

1. Introduction

The theory of computational complexity (the part of computer theory that studies the resources required during the calculation to solve a problem [1]) introduces the notion of complexity classes (*P*, *NP*, *NP*-complete and *NP*-hard) of real problems. *P* problems are the class of problems that can be solved in polynomial time as a function of the given input size [2]. *NP* problems are the set of problems that can be verified in polynomial time as a function of the given input size [2] using a nondeterministic Turing machine. *NP*-complete problems constitute the class of the most difficult possible *NP* problems.

This paper aims at being a guide to understand polynomial transformations between *NP*-complete problems. These kinds of transformations have been used for: (a) proving that a problem is easy by reducing it to a problem known to be solvable by a polynomial algorithm, for example Ford and Fulkerson in 1962 [3] described many of these reductions; (b) proving that a problem is difficult through reduction to show that some problems are difficult, some examples of reductions were described by Dantzig, Blattner and Rao in 1967 [4]; (c) reducing a problem of unknown complexity to another problem that is or is suspected to be difficult. A polynomial transformation between *NP*-complete problems allows a language (L_1) and a language (L_2) to be transformed in polynomial time.

Section 2 describes the methodologies of polynomial transformations. In Section 3 the historical transformations between *NP*-complete problems are presented. Section 4 explains the differences between a polynomial reduction and a polynomial transformation. Finally, in Section 5 the conclusions and some remarks of this research are presented.

* Corresponding author. Tel.: +52 7771591325.
E-mail addresses: jorge@ruizvanoye.com (J.A. Ruiz-Vanoye), jpo_cenidet@yahoo.com.mx (J. Pérez-Ortega), r_pazos_r@yahoo.com.mx (R.A. Pazos R.), ocotlan@diazparra.net (O. Díaz-Parra), juan.frausto@itesm.mx (J. Frausto-Solís), hfraire@prodigy.net.mx (H.J. Fraire Huacuja), lauracruzreyes@hotmail.com (L. Cruz-Reyes), jose.mtz@gmail.com (J.A. Martínez F.).

0377-0427/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.cam.2011.02.018

2. Methodologies of polynomial transformations

In this section, we show the methodologies of polynomial transformation between *NP*-complete problems using several examples:

(a) For polynomial reductions. Knapsack Problem $\propto$ 2-Partition Problem (Section 2.1.1), SAT $\propto$ Clique (Section 2.1.2), and SAT $\propto$ 0–1 Integer Programming (Section 2.1.3).
(b) For polynomial transformations. HC $\leq_P$ TSP (Section 2.2.1), SAT $\leq_P$ 3-CNF SAT (Section 2.2.2), 1D-BPP $\leq_P$ 2-PAR (Section 2.2.3).

2.1. Polynomial reductions

Post [5] and later Ladner et al. [6,7] introduced the notion of truth-table reducibility. A set A is truth reducible to B ($A \leq_{tt} B$) if, given x, one can effectively compute a finite set of words, say $y_1, \ldots, y_k$, and a Boolean function $\propto$ such that $x \in A$ if and only if $\propto (C_B(y_1), \ldots, C_B(y_k)) = 1$, where $\propto$ was represented by its table.

The term polynomial reducibility was previously denominated in different ways: P-reducibility was the basic Cook's notion of reducibility between languages, which contained polynomial time Turing reducibility, occasionally known as Cook reducibility [1]. A set A is Turing reducible to a set B in polynomial time ($A \leq_T^P B$) if there is an oracle Turing machine M and a polynomial function p such that $x \in A$ iff M accepts x with B as its oracle within $p(n)$ steps. (An oracle Turing machine is a multi-tape Turing machine with a special oracle tape and special states Q, YES, and NO. Should the machine enter state Q, then the next state is YES or NO depending on whether or not the string currently written on the oracle tape is in the oracle set.)

Later Meyer and Stockmeyer [8] have used a notion of nondeterministic polynomial time Turing reducibility. They proposed three types of reducibility: (a) A is nondeterministic polynomial time Turing reducible to B ($A \leq_T^{NP} B$) if and only if there is a nondeterministic oracle Turing machine M that runs in polynomial time, and A is accepted by M with oracle B [6,7]. (b) A is nondeterministic polynomial time many-one reducible to B ($A \leq_m^{NP} B$) if and only if there is a nondeterministic Turing machine transducer M that runs in polynomial time such that $x \in A$ just in case there is a y computed by M on input x with $y \in B$ [6,7]. (c) A is a nondeterministic polynomial time truth table reducible to B ($A \leq_{tt}^{NP} B$) if and only if there is a nondeterministic Turing machine transducer M that runs in polynomial time and a polynomial time computable evaluator e such that $x \in A$ just in case on input x, M computes a tt-condition y which is e-satisfied by B [6,7].

In the same year, Karp [9] changed the original term from polynomial Turing reducibility to polynomial transformability and called it *Karp reducibility*, which permits us to reduce the members of the problem class among them, occasionally called polynomial time bounded many-one reducibility [1,2,9]. A set A is many-one reducible to B in polynomial time ($A \leq_m^P B$) if there is a function f computable in polynomial time such that $x \in A$ if and only if $f(x) \in B$. Karp used the notation $\propto$ instead of $\leq_m^P$. Finally, polynomial reductions between A and B are frequently denoted by $A \propto B$ or $A \leq_T^P B$, or incorrectly by $A \leq_P B$, incorrectly because the symbol $\leq_P$ is used for a polynomial transformation too.

The most famous first *polynomial transformations* of instances known as *polynomial reductions* are attributed to Karp [9] and Cook [1], although initially they considered language reduction from a problem to another problem. It is necessary to mention that what they considered as language was in fact chains of symbols and not properly a formal language as is known nowadays.

Definition of polynomial reduction (Karp) [9]: Let L and M be two languages, then L is reducible to M if there exists a function $f \in M \Leftrightarrow x \in L$. Incidentally, for Karp a subset of $\sum^*$ (the set of all the finite chains of 0s and 1s) is called language.

Definition of polynomial reduction (Cook) [1]: A set S of chains of symbols (on a fixed, large and finite alphabet $\{0.1, ^*\}$) is polynomial reducible to a set T of chains of symbols (on a fixed, large and finite alphabet $\{0.1, ^*\}$) if there exists a query machine M and a polynomial $Q(n)$, such that for each input string w the computation time of M with input w halts in $Q(|w|)$ steps ($|w|$ is the length of w) and ends in the accept state iff $w \in S$. Cook [1] defined a language as a set G of chains of symbols on a fixed, large and finite alphabet $\{0, 1, ^*\}$.

In other words, Cook's reduction implied the analysis of an input string using a one-tape Turing machine (with rules for state transitions) and one query tape (that contained a finite list of the valid strings for that Turing machine and its equivalence in the other language).

Karp mentions 21 polynomial reductions between *NP*-complete problems [9]. In subsequent subsections we exemplify the following polynomial reductions: Knapsack Problem $\propto$ 2-Partition Problem, SAT $\propto$ Clique, and SAT $\propto$ 0–1 Integer Programming.

2.1.1. Example of reduction Knapsack Problem $\propto$ 2-partition

The polynomial reduction will be carried out from a language for the Knapsack Problem to another language for the 2-Partition Problem.

Definition of the Knapsack Problem [9,10]: Given a set of objects of sizes a_j ($j = 1, \ldots, r$) and a vector of binary variables x_j ($j = 1, \ldots, r$) with value 1 if object j is selected and 0 otherwise, and a knapsack of capacity b. Then, if p_j is a measure of the comfort given by object j, the problem consists of selecting objects such that the overall sum of their sizes does not

exceed the knapsack capacity. The input parameters of the Knapsack Problem are: $K = \{a_1, a_2, \ldots, a_r, b\}$, where b is the capacity of the knapsack and a_j are the objects to be assigned to the knapsack.

Definition of the 2-Partition Problem [9,10]: Given a set of integer numbers U, the problem consists of determining if there exists a partition constituted by two disjoint subsets A and A^c, such that each number is assigned to just one subset and the sum of the numbers in subset A equals that of subset A^c. The input parameters of 2-Partition are: $P = \{c_1, c_2, \ldots, c_s\}$, where the c_i's are integer numbers.

Expressions (1)–(4) were proposed by Karp [9] for reducing the Knapsack Problem to 2-Partition.

$$s = r + 2 \tag{1}$$

$$c_i = a_i \quad (i = 1, \ldots, r) \tag{2}$$

$$c_{r+1} = b + 1 \tag{3}$$

$$c_{r+2} = \sum_{i=1}^{r} a_i + 1 - b. \tag{4}$$

In this example we reduce the Knapsack instance $K = (1, 1, 1, 1, 3)$ to an instance of 2-Partition.

Step 1. Determine first the values of r (the number of objects) and b, in this case $r = 4$ and $b = 3$.

Step 2. Use expression (1) for starting the reduction process between the two problems, thus obtaining $s = 4+2 = 6$, which means that the vector for 2-Partition will have 6 items, $P = (\_, \_, \_, \_, \_, \_)$.

Step 3. Use expression 2 for reducing the elements of vector a of the Knapsack Problem to vector c elements of 2-Partition. The reduction yields $P = (1, 1, 1, 1, \_, \_)$, remaining two elements c_{r+1} and c_{r+2} of 2-Partition without values.

Step 4. Calculate the value of c_{r+1} using expression (3), which yields $c_{r+1} = b + 1 = 3 + 1 = 4$, so the vector for 2-Partition would be $P = (1, 1, 1, 1, 4, \_)$, and the element c_{r+2} of 2-Partition would remain without value.

Step 5. Calculate the value of c_{r+2} using expression (4), thus yielding $c_{r+2} = (1 + 1 + 1 + 1) + 1 - 3 = 2$; thus, the resulting vector for 2-Partition would be $P = (1, 1, 1, 1, 4, 2)$.

Step 6. Finally, we obtained the vector for 2-Partition reduced from the Knapsack Problem, which can be expressed as follows: $K = (1, 1, 1, 1, 3) \leq_P P = (1, 1, 1, 1, 4, 2)$.

2.1.2. Example of reduction SAT $\propto$ clique

The polynomial reduction will be carried out from a language for SAT to another language for the Clique problem.

Definition of the Satisfiability (SAT) problem [9]: Given a Boolean formula, find an assignment of values to each of its variables such that the formula evaluates to 1 or verifies that there does not exist such an assignment. The input parameters of SAT are the set of clauses $C = \{c_1, c_2, \ldots, c_p\}$, and variables $X = \{x_1, x_2, \ldots, x_n\}$.

Definition of the Clique problem [2,9]: Given a graph and an integer number k, does there exist k vertices that are adjacent to each other? The input parameters of Clique are a graph $G = (V, A)$ and a number k of vertices.

Expressions (5)–(7) [9] permit us to polynomially reduce from SAT to Clique.

$$k = p, \quad \text{is the number of clauses} \tag{5}$$

$$V = \{\langle x', i\rangle; x' \text{ is a variable or its negation and occurs in } c_i\} \tag{6}$$

$$A = \{\{\langle x', i\rangle, \langle y', j\rangle\}; i \neq j \text{ and } x' \neq \neg y'\}. \tag{7}$$

In this example we reduce the SAT instance $(x_1 \vee x_2 \vee x_3 \vee \neg x_4)(\neg x_3 \vee x_4 \vee \neg x_5)$ to a Clique instance.

Step 1. Use expression (5) for determining one of the Clique dimensions, in this case $k = p = 2$.

Step 2. Use expression (6) for determining the Clique vertices:

$$V = \{\langle x_1, 1\rangle, \langle x_2, 1\rangle, \langle x_3, 1\rangle, \langle \neg x_4, 1\rangle, \langle \neg x_3, 2\rangle, \langle x_4, 2\rangle, \langle \neg x_5, 2\rangle\}.$$

Step 3. Use expression (7) for determining the Clique arcs; thus

$$\begin{aligned} A = {} & \{\{\langle x_1, 1\rangle, \langle x_3, 2\rangle\}, \{\langle x_1, 1\rangle, \langle x_4, 2\rangle\}, \{\langle x_1, 1\rangle, \langle \neg x_5, 2\rangle\}, \{\langle x_2, 1\rangle, \langle \neg x_3, 2\rangle\}, \\ & \{\langle x_2, 1\rangle, \langle x_4, 2\rangle\}, \{\langle x_2, 1\rangle, \neg\langle x_5, 2\rangle\}, \{\langle x_3, 1\rangle, \langle x_4, 2\rangle\}, \{\langle x_3, 1\rangle, \neg\langle x_5, 2\rangle\}, \\ & \{\langle \neg x_4, 1\rangle, \neg\langle x_3, 2\rangle\}, \{\langle \neg x_4, 1\rangle, \neg\langle x_5, 2\rangle\}\}. \end{aligned}$$

Step 4. The resulting Clique would look as depicted in Fig. 1.

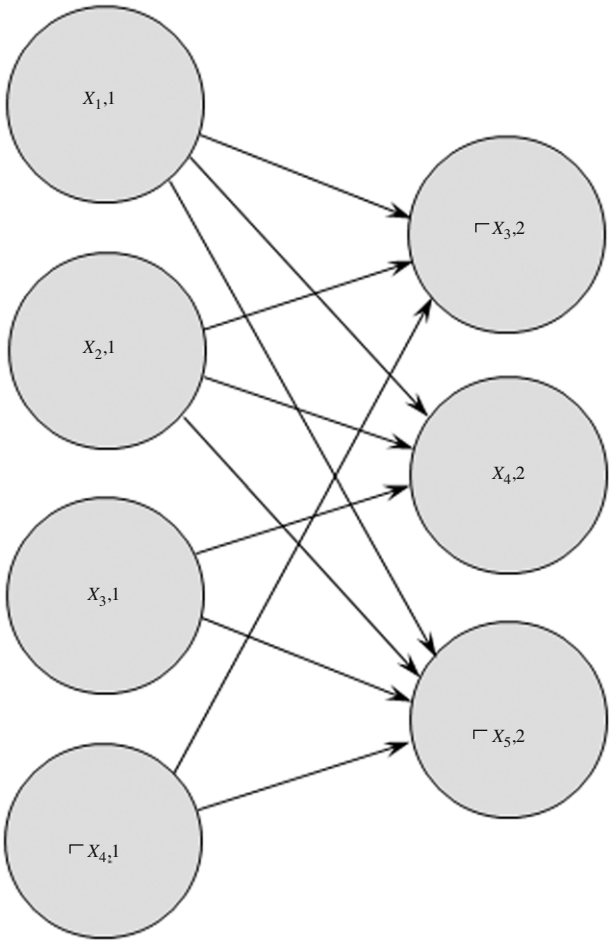

Fig. 1. Instance of the Clique problem.

$$\begin{bmatrix} 1 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 \end{bmatrix} \mathrm{x} \begin{bmatrix} ? \\ ? \\ ? \\ ? \\ ? \end{bmatrix} \geq \begin{bmatrix} -1 \\ -1 \end{bmatrix}$$

Fig. 2. Example of 0–1 Integer Programming Problem.

2.1.3. Example of reduction SAT $\propto$ 0–1 INT

The polynomial reduction is performed from a language for SAT to another language for the 0–1 Integer Programming Problem.

Definition of the Satisfiability problem [9]: same as in Section 2.1.2.

Definition of the 0–1 Integer Programming Problem (0–1 INT) [9]: Given a matrix A and a vector b, does there exist a vector x with values $\{0, 1\}$ such that $Ax \geq b$? An example of 0–1 Integer Programming Problem is in Fig. 2. The 0–1 INT input parameters are: an integer matrix M and a vector b.

Expressions (8) and (9), proposed in [9], are used for reducing from SAT to 0–1 INT.

$$m_{ij} = \begin{cases} 1, & \text{if } x_j \in c_i;\ i = 1, 2, \ldots, p \\ -1, & \text{if } \neg x_j \in c_i;\ j = 1, 2, \ldots, n \\ 0, & \text{otherwise} \end{cases} \tag{8}$$

$$b_i = 1 - (\text{number of complemented variables in } c_i); \quad i = 1, 2, \ldots, p. \tag{9}$$

In this example we will reduce the SAT instance $(x_1 \vee \neg x_2 \vee \neg x_3)(\neg x_4 \vee \neg x_5)$ to a 0–1 INT instance.

Step 1. First identify the number of rows of the matrix with the number of clauses $= p = 2$; then identify the number of columns of the matrix with the number of variables $= n = 5$; as a result, we obtain a 2×5 matrix M and a vector b of size 2.

Step 2. Next find out if the variables are present in clause 1, and assign a 1 if x is present, a -1 if $\neg x$ is present, and 0 if it is not present. These values are assigned to the first row of matrix M.

$$\begin{bmatrix} 1 & -1 & -1 & 0 & 0 \\ - & - & - & - & - \end{bmatrix}.$$

Step 3. Find out if the variables are present in clause 2, and make an assignment of values to the second row of matrix M, applying a process similar to that for the first row.

$$\begin{bmatrix} 1 & -1 & -1 & 0 & 0 \\ 0 & 0 & 0 & -1 & -1 \end{bmatrix}.$$

Step 4. Determine the values for vector b, which can be obtained from the reduction expressions. The first value of the vector can be determined by finding out how many $\neg x$'s are present in the first clause, which in this case is 2; therefore, $b_1 = 1 - 2 = -1$. The resulting vector looks as shown below.

$$\begin{bmatrix} -1 \\ - \end{bmatrix}.$$

Step 5. Determine the second value of the vector by applying a process similar to the one explained for the first value; therefore, $b_1 = 1 - 2 = -1$. The final vector looks as shown below.

$$\begin{bmatrix} -1 \\ -1 \end{bmatrix}.$$

Step 6. Finally, matrix M and vector b (input parameters of 0–1 INT, transformed from SAT) are obtained.

$$(x_1 \vee \neg x_2 \vee \neg x_3)(\neg x_4 \vee \neg x_5) \leq_P \begin{bmatrix} 1 & -1 & -1 & 0 & 0 \\ 0 & 0 & 0 & -1 & -1 \end{bmatrix} \begin{bmatrix} -1 \\ -1 \end{bmatrix}.$$

2.2. Polynomial transformations

There exist different approaches for carrying out transformations: using the theory of *NP*-completeness [2], using graph theory [11], and using formal language theory [12]. Though these approaches aim at the same objective, they are different in certain aspects.

The *NP*-completeness approach consists of four steps for reducing *NP*-complete problems [2]:

1. Show that problem B is in *NP*, i.e., $B \in NP$.
2. Select a problem A, known to be in the *NP*-complete class. (Note: it is convenient to select a problem A that is similar to B.)
3. Devise a transformation algorithm f from problem A to problem B.
4. Verify that f is a polynomial transformation function.

For performing transformations using graph theory, the following steps should be carried out [11]:

1. Select a problem B which is known to be *NP*.
2. Select a problem A, known to be in the *NP*-complete class, and construct a graph for decision problem A.
3. Devise a transformation algorithm from problem A to problem B.
4. Verify the transformation algorithm.
5. Construct a graph for decision problem A using the transformation algorithm.

Unlike most transformations, which are used for proving that a problem is *NP*-complete based on the *NP*-completeness of another problem, the proposed approach (using formal language theory [13]) is intended for extrapolating some known characteristics, phenomenon or behavior from a problem A to another problem B. The steps for performing a transformation using formal language theory are [12]:

1. Select an *NP*-complete problem A (source problem) for polynomial transformation.
2. Define a formal language L_1 (source language) for the *NP*-complete problem A.
3. Select an *NP*-complete problem B (target problem).
4. Define a formal language L_2 (target language) for problem B.
5. Construct a compiler that transforms in polynomial time a source language into a target language.
6. Once the formal language L_1 for problem A and the formal language L_2 for problem B are defined, the compiler has to be used on the source language (L_1) for obtaining the target language (L_2). This compilation constitutes the polynomial transformation from problem A to problem B ($A \leq_P B$).

In subsequent subsections, the following polynomial transformations will be exemplified: HC $\leq_P$ TSP, SAT $\leq_P$ 3-CNF SAT, 1D-BPP $\leq_P$ 2-PAR.

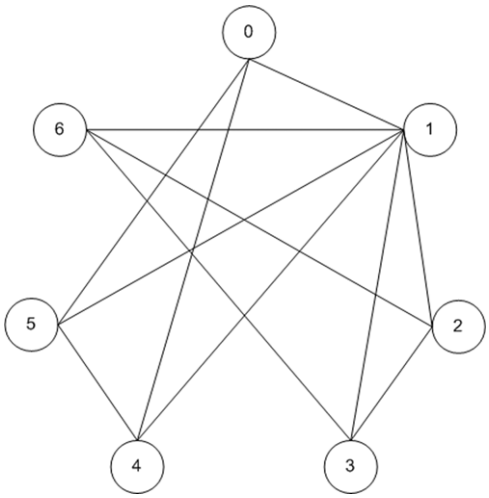

Fig. 3. Graph of the HC instance with 7 vertices.

2.2.1. Example of transformation HC $\leq_P$ TSP

The polynomial transformation using graph theory that is exemplified, is from the Hamiltonian Circuit (HC) to the Traveling Salesman Problem (TSP).

Definition of the Traveling Salesman Problem [2]: Given a set of nodes and distances for each pair of nodes, find a circuit of minimal overall length that visits each of the nodes exactly once [2]. The distance from node *i* to node *j* and the distance from node *j* to node *i* can be different. In other words, the problem consists of finding the shortest circuit in a weighted graph or for a set of points on a plane.

Definition of the Hamiltonian Circuit problem [2]: Given a graph, find a Hamiltonian Circuit (a circuit that visits each of the nodes exactly once).

In order to make transformations using graph theory, the following steps have to be applied:

Step 1. Show that problem *B* belongs to the class of *NP* problems. For example, TSP $\in$ *NP*. For showing this the polynomial reduction for HC (Directed Hamiltonian path or Undirected Hamiltonian path) $\propto$ TSP is used.

Step 2. Select a problem *A* that is known to be *NP*-complete and create a graph for the decision problem. For example (Fig. 3), the graph $G = (V, E)$ for an HC instance with 7 vertices is selected. Does graph *G* contain a Hamiltonian Circuit $\langle v_1, v_2, \ldots, v_n \rangle$ of the vertices of *G*?

Step 3. Create a transformation algorithm. In this case the algorithm for the transformation of the HC instance to the TSP instance [11] is the following:

numberVertices = *getNumberVertices (Hamilton Graph)*
tspGraph = *createCompleteGraph(numberVertices)*
foreach *edge in tspGraph* ***do***
if *containsEdge(edge, hamiltonGraph)* ***then***
setEdgeWeight (edge, 1*)*
else *setEdgeWeight(edge, numberVertices* +1)
end if
end foreach
return (*G*, *numberVertices*)

The transformation algorithm in this example first creates a fully connected graph with the same number of vertices as the HC instance. Then it assigns a weight to each edge: 1 if the HC instance contains the same edge, and $n + 1$ otherwise (n is the number of vertices in the graph). To this end, the Java API for graphs contained in GraphBench can be used [11].

Step 4. Verify (Fig. 4) the transformation algorithm f (*instance I*) $= (G, k)$. An HC instance $G = (V, E)$ transforms into a TSP instance $G' = (V, E')$, where $E' = \{(i, j) : i, j \in V \text{ and } i \neq j\}$, where one has a function f that transforms an instance I into an input for TSP, a graph G and an integer number *numberVertices* or k.

In order to determine that problem *A* is transformable to problem *B*, it is necessary to apply a subroutine to problem *A* and verify that it obtains the same solution (yes or no) that it obtains when applied to problem *B* for each instance transformed.

foreach *edge* (u, v) *of G// of HC* ***do***
if *there is a path of length* $n - 1$ *from u to v*
return *yes*
return *no*

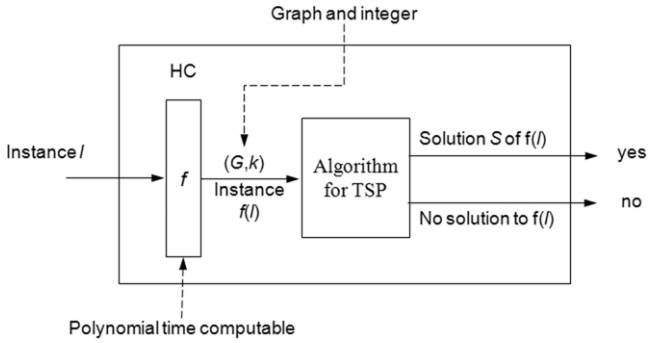

Fig. 4. Verification scheme for reducing HC to TSP.

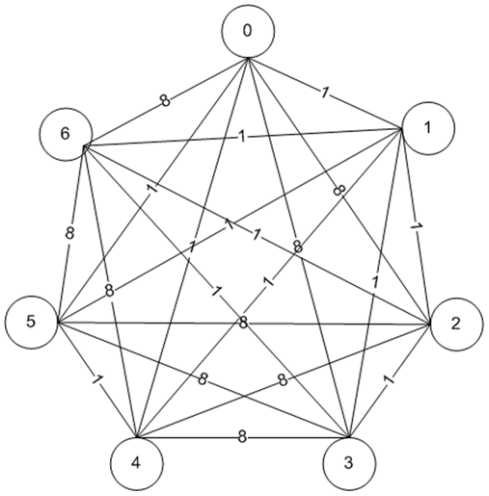

Fig. 5. Graph resulting from applying the transformation algorithm [11].

foreach *edge* (u, v) *of* $G//$ *of TSP* ***do***
if *there is a path of length* $n-1$ *from* u *to* v
return *yes*
return *no*

Step 5. Create a graph for decision problem B using the transformation algorithm. Example, the resulting graph of the transformation algorithm HC $\leq_P$ TSP is the graph of a TSP instance (depicted in Fig. 5).

2.2.2. Example of transformation SAT $\leq_P$ 3-*CNF SAT*

The polynomial transformation using Garey and Johnson's theory is exemplified by the transformation from the Satisfiability problem to the 3-CNF Satisfiability.

Definition of the Satisfiability problem [9]: same as in Section 2.1.2.

Definition of the 3-CNF Satisfiability (3-SAT) problem [9]. Consider a collection $C = \{c_1, c_2, \ldots, c_m\}$ of clauses of a finite set U of n Boolean variables. Let (the literal) u be true if and only if variable u is true and let $\neg u$ be true iff u is false. Each clause c_i contains a disjunction of three literals over U. A collection C of clauses over U is satisfiable if and only if there exists some truth assignment for U that simultaneously satisfies all the clauses in C. The 3-SAT decision problem consists of determining whether there is a truth assignment for U that satisfies all the clauses in C.

For carrying out the polynomial transformation SAT $\leq_P$ 3-SAT, the approach proposed by Garey and Johnson is used.

Step 1. Prove that 3-SAT is in *NP*, i.e., 3-SAT $\in$ *NP*. To this end it is necessary to devise a nondeterministic algorithm that guesses a truth assignment for the variables and checks in polynomial time whether this truth setting satisfies all the three-literal clauses.

Step 2. Select a problem A which is known to be *NP*-complete. For our example, it is known that SAT is *NP*-complete; incidentally, its proof is attributed to Cook [1].

The instance selected for A is $(x1 \vee x2) \wedge (\neg x1 \vee x2) \wedge (x1 \vee \neg x2) \wedge (\neg x1 \vee \neg x2)$.

Step 3. Construct a transformation algorithm f from problem A (SAT) to problem B (3-SAT). The algorithm takes the original formula with its variables $X = \{x_1, x_2, \ldots, x_n\}$ and its clauses $C = \{c_1, c_2, \ldots, c_m\}$ and generates a new formula for 3-SAT.

The original formula is introduced into a set of clauses and variables depending on the number k of literals (variables or their negations) of the original clause.

If $k = 1$: originalClause: $c_j = (l_1)$; AdditionalVariables: y_j^1, y_j^2;

NewClauses: $(l_1 \vee y_j^1 \vee y_j^2) \wedge (l_1 \vee y_j^1 \vee \neg y_j^2) \wedge (l_1 \vee \neg y_j^1 \vee y_j^2) \wedge (l_1 \vee \neg y_j^1 \vee \neg y_j^2)$.

If $k = 2$: originalClause: $c_j = (l_1 \vee l_2)$; AdditionalVariables: y_j^1;

NewClauses: $(l_1 \vee l_2 \vee y_j^1) \wedge (l_1 \vee l_2 \vee \neg y_j^1)$.

If $k = 3$: originalClause: $c_j = (l_1 \vee l_2 \vee l_3)$;

AdditionalVariables: No-AdditionalVariables; NewClauses: $(l_1 \vee l_2 \vee l_3)$.

If $k > 3$: originalClause: $c_j = (l_1 \vee l_2 \vee \cdots \vee l_k)$;

AdditionalVariables: $y_j^1, y_j^2, \ldots, y_j^{k-3}$;

NewClauses: $(l_1 \vee l_2 \vee y_j^1) \wedge (\neg y_j^1 \vee l_3 \vee y_j^2) \wedge (\neg y_j^2 \vee l_4 \vee y_j^3) \wedge \cdots \wedge (\neg y_j^{k-4} \vee l_{k-2} \vee y_j^{k-3}) \wedge (\neg y_j^{k-3} \vee l_{k-1} \vee l_k)$.

The transformation algorithm (the Java API for graphs contained in GraphBench [11] can be used) from SAT to 3-SAT is the following:

Copyvariables(satisfiabilityFormula, 3CNFFormula)
foreach *originalClause in satisfiabilityFormula* ***do***
clauseSize = sizeof(originalClause)
if *clauseSize* = 1 ***then***
*addAdditionalVariables(*2, 3*CNFFormula)*
addFourClauses(originalClause, 3*CNFFormula)*
else if *clauseSize* = 2 ***then***
*addAdditionalVariables(*1, 3*CNFFormula)*
addTwoClauses(originalClause, 3*CNFFormula)*
else if *clauseSize* = 3 ***then***
addClause(originalClause, 3*CNFFormula)*
else
addAdditionalVariables(clauseSize −2, 3*CNFFormula)*
addClauseChain(originalClause, 3*CNFFormula)*
end if
end foreach.

Step 4. Verify that f is a transformation function.

After applying the algorithm, an instance of 3-SAT was obtained:

$$(x1 \vee x2 \vee y11) \wedge (x1 \vee x2 \vee \neg y11) \wedge (\neg x1 \vee x2 \vee y21) \wedge (\neg x1 \vee x2 \vee \neg y21)$$
$$\wedge (x1 \vee \neg x2 \vee y31) \wedge (x1 \vee \neg x2 \vee \neg y31) \wedge (\neg x1 \vee \neg x2 \vee y41) \wedge (\neg x1 \vee \neg x2 \vee \neg y41).$$

We use a physical model for trying to solve SAT and 3-SAT using physical forces. Every clause pulls or pushes with a certain force on "its" variables, trying to make them adopt a value that satisfies the clause. The closer a variable is to a desired value, the stronger a clause affects the variable. To avoid being stuck in a local minimum, a small random value is added to the forces. In each iteration step the forces applied on all variables are first calculated. These values are then used to calculate new variable values. This is done until the formula is satisfied. For example consider $c_1 = (x1 \vee \neg x2 \vee x3)$, for clause c_1 to be satisfied $x1$ has to be true, $x2$ false or $x3$ true. The c_1 therefore tries to push $x1$ and $x2$ to 1 and tries to pull $x2$ towards 0. With this, it is determined that both are yes-instances of SAT and 3-SAT.

2.2.3. Example of transformation 1D-BPP $\leq_P$ 2-PAR

The polynomial transformation using formal language theory is exemplified by the transformation from the one-dimensional Bin Packing Problem to the 2-Partition Problem.

Definition of the one-dimensional Bin Packing Problem (1D-BPP) [10]: given a finite set U of n items with sizes $w = \{w_1, \ldots, w_n\}$, a positive integer number c that represents the bin capacity and a positive integer number K (maximal number of bins), the problem consists of determining if there exists a partition of U consisting of disjoint sets $U_1, U_2, \ldots, U_K$ such that the sum of the items sizes in each U_i is c or less.

Definition of the 2-Partition Problem [10]: same as in Section 2.1.1.

The steps for the transformation from 1D-BPP to 2-Partition using formal language theory are the following:

Step 1. Select an *NP*-complete problem A (source problem) for its polynomial transformation. In this case 1D-BPP was selected since it is known to be *NP*-complete, i.e., 1D-BPP $\in$ *NP*-complete [11].

Step 2. Define a formal language L_1 (source language) for the *NP*-complete problem A. For example, for 1D-BPP, the alphabet $\sum$ = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, "{", "}", ",", ";", " = ", "B", "P", "D", " − "} and a BNF (Backus–Naur Form) grammar were defined. For example, *pname* = {*num*; $w_1, w_2, \ldots, w_{num}$; c; k}, where *num* = overall number of items, w_i = size of the item i, c = bin capacity, k = number of bins, *pname* = the name of the problem; therefore, for this example L_1 = {1D-BPP = {6; 10, 20, 20, 10, 5, 5; 20; 4}}.

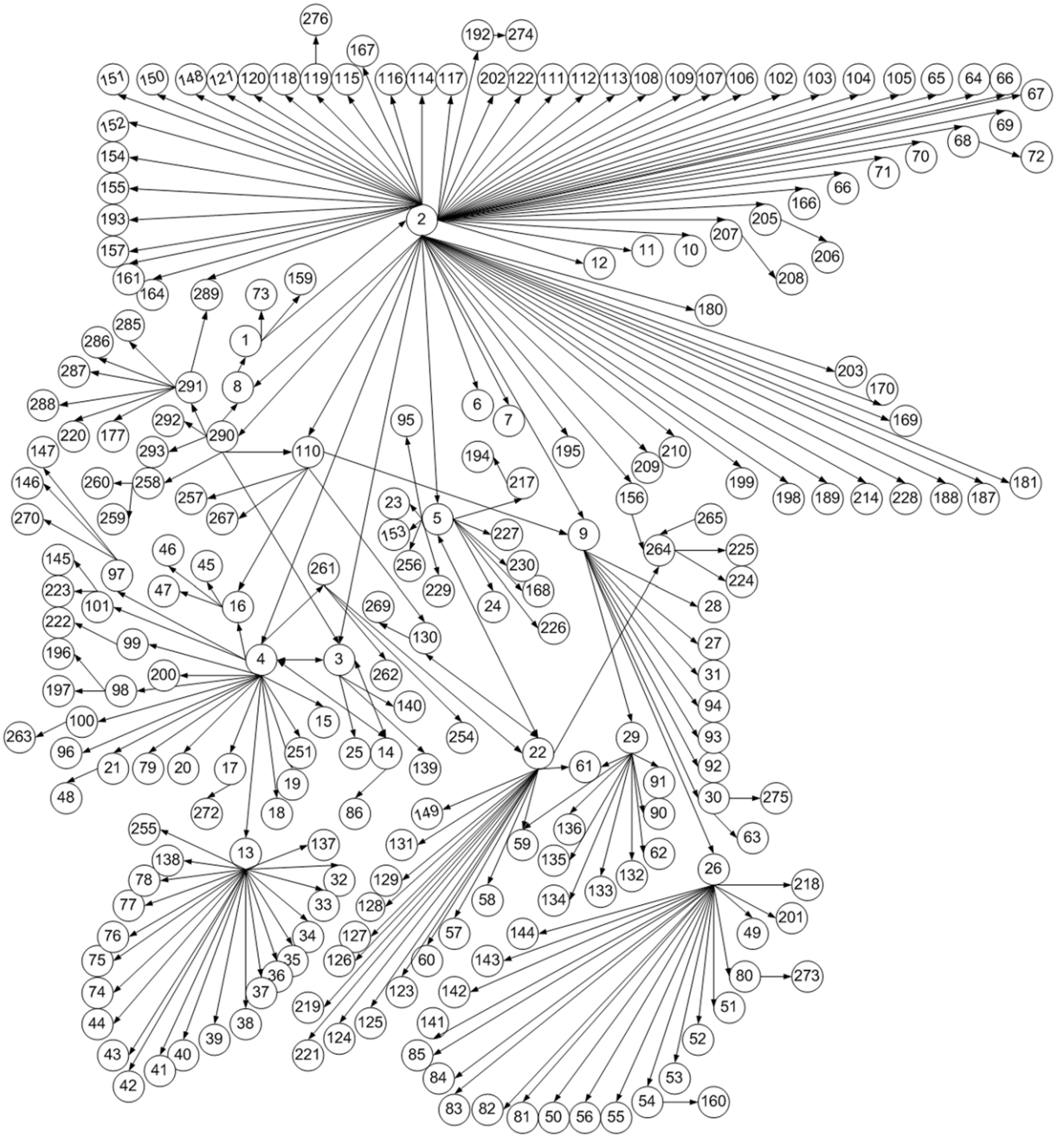

Fig. 6. Digraph of transformations between *NP*-complete problems.

Step 3. Select an *NP*-complete problem *B* (target problem). In this case 2-Partition was selected since it is known to be *NP*-complete [2,10].

Step 4. Define a formal language L_2 (target language) for the *NP*-complete problem *B*. In this case, the alphabet $\sum$ = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, "{", "}", ", ", "; ", " = ", "*A*", "*P*", "*R*", " − "} and a BNF grammar were defined. For example, L_2 = {2-PAR = {*num*; $c_1, c_2, \ldots, c_{num}$}}.

Step 5. Construct a compiler (a computer program or set of programs that translates text written in a source language into another target language, such as Lex, Flex, Yacc, or Bishop) that performs in polynomial time the transformation 1D-BPP $\leq_P$ 2-PAR; i.e., use a compiler that takes as input a source language (L_1) for obtaining a target language (L_2), in order to carry out the polynomial transformation from problem *A* to problem *B* ($A \leq_P B$). For this example, as a result of the polynomial transformation using formal languages, 2-PAR = {6; 10, 20, 20, 10, 5, 5} was obtained.

Table 1
List of problems (part 1).

Number	Name of problem	Number	Name of problem
1	Satisfiability (SAT)	2	3-Satisfiability (3SAT)
3	Clique (clique cover)	4	Vertex cover
5	Subset sum	6	Hitting string
7	Chinese postman for mixed graphs	8	Graph colorability
9	Three-Dimensional matching (3DM)	10	Rectilinear picture compression
11	Tableau equivalence	12	Consistency of databases frequency tables
13	Hamiltonian Circuit (Directed Hamiltonian path, Undirected Hamiltonian path)	14	Independent set
15	Setbasis	16	Hitting set
17	Comparative containment	18	Multiple copy file allocation
19	Shortest common supersequence	20	Longest common subsequence
21	Minimum cardinality key	22	Partition
23	*K*th largest subset	24	Capacity assignment
25	Conjunctive Boolean query	26	Exact cover by 3-sets (X3C)
27	Minimum test set	28	3-Matroid intersection
29	3-Partition	30	Numerical three-dimensional matching
31	Pruned trie space minimization	32	Consecutive sets
33	Traveling Salesman Problem	34	Longest path
35	Longest circuit	36	Rural postman
37	Stacker crane	38	Bottleneck TSP
39	Consecutive ones submatrix	40	Consecutive ones matrix partition
41	Consecutive block minimization	42	Degree constrained spanning tree
43	Internal macrodata compression	44	External macrodata compression
45	Boyce–Codd normal form violation	46	Additional key
47	Safety of database transaction Systems	48	Prime attribute name
49	Shortest total path length spanning tree	50	Bounded diameter spanning tree
51	Regular expression substitution	52	Geometric TSP
53	Set packing	54	Minimum cover
55	Subset product	56	Expected component sum
57	Expected retrieval cost	58	*K*th largest *m*-tuple
59	Bin packing	60	Shortest weight constrained path
61	Minimum sum of squares	62	Dynamic storage allocation
63	Numerical matching with target sums	64	Capacitated spanning tree
65	Multiple choice branching	66	Path constrained network flow
67	Integral flow with homologous Arcs	68	Directed two-commodity integral flow
69	Disjoint connecting paths	70	Maximum Length Bounded disjoint paths
71	Maximum fixed-length disjoint paths	72	Undirected two-commodity integral flow
73	Undirected flow with lower bounds	74	Isomorphic spanning tree
75	Quadratic assignment problem	76	Strong connectivity augmentation
77	Biconnectivity augmentation	78	*K*th spanning tree
79	Minimizing dummy activities in pert networks	80	Steiner tree in graphs
81	Geometric Steiner tree	82	Acyclic partition
83	Minimum edge-cost flow	84	Geometric capacitated spanning tree
85	Optimum communication spanning tree	86	Integral flow with bundles
87	Bounded component spanning forest	88	Graph partitioning
89	Integral flow with multipliers	90	Intersection graph for segments on a grid
91	Edge embedding on a grid	92	Minimum broadcast time
93	Partition into triangles	94	Partition into isomorphic subgraphs
95	Integer knapsack	96	Scheduling with individual deadlines
97	Dominating set	98	Feedback vertex set
99	Feedback arc set	100	Partial feedback edge set

Notice that the example transformation 1D-BPP to 2-PAR was carried out inversely as is usually performed. The transformation from 2-PAR to 1D-BPP is usually realized for proving that 1D-BPP is *NP*-complete assuming that 2-PAR is *NP*-complete. However, in our example we actually carried out this transformation for obtaining indicators that predicted the performance of optimization algorithms applied to 2-PAR based on the previously known performance of these algorithms when applied to 1D-BPP.

3. Historical transformations between *NP*-complete problems

In the specialized literature there exist various approaches for carrying out transformations between instances of *NP*-complete problems. Fig. 6 depicts a digraph with the historical reductions/transformations between instances of *NP*-complete problems, where these problems (Tables 1–4) are represented by circles with a number inside. The digraph shows transformations represented by the direction of an arrow, for example: if there exists an arrow from problem *A* to problem *B*, it means that *A* transforms to *B* ($A \leq_P B$), and if there exists a double headed arrow between problem *A* and problem *B*, it means that *A* transforms to *B* and vice versa ($A \leq_P \geq B$).

Table 2
List of problems (part 2).

Number	Name of problem	Number	Name of problem
101	Minimum maximal matching	102	Traveling salesman polytope non-adjacency
103	Cost-parametric linear programming	104	Integer programming
105	Deadlock avoidance	106	Timetable design
107	Preemptive scheduling	108	Precedence constrained scheduling
109	Domatic number	110	Exact cover
111	Monochromatic triangle	112	Partition into Hamiltonian subgraphs
113	Quadratic congruences	114	Simultaneous incongruences
115	Comparative divisibility	116	Exponential expression divisibility
117	Non-divisibility of a product polynomial	118	Non-trivial greatest common divisor
119	Quadratic diophantine equations	120	Root of modulus 1
121	Number of roots for a product polynomial	122	Periodic solution recurrence relation
123	Multiprocessor scheduling	124	Sequencing with deadlines and set-up times
125	Sequencing to minimize tardy task weight	126	Scheduling to minimize weighted completion time
127	Open-shop scheduling	128	Production planning
129	Quadratic programming	130	Knapsack
131	Continuous multiple choice knapsack	132	Sequencing with release times and dead lines
133	Sequencing to minimize weighted tardiness	134	Job-shop scheduling
135	Resource constrained scheduling	136	Flow-shop scheduling
137	No wait flow-shop scheduling	138	Feasible basis extension
139	Sequencing to minimize tardy task	140	Partially ordered knapsack
141	Staff scheduling	142	Minimum weight solution to linear equations
143	K-relevancy	144	Algebraic equations over GF
145	Achromatic number	146	Min–max multicenter
147	Min-sum multicenter	148	Permanent evaluation
149	Cosine product integration	150	Equilibrium point
151	Unification with commutative operators	152	Unification for finitely presented algebras
153	Integer expression membership	154	Not-all-equal 3SAT
155	One-in-three 3SAT	156	Maximum 2-Satisfiability
157	Generalized satisfiability	158	Satisfiability of Boolean expressions
159	Non-tautology	160	Minimum disjunctive normal form
161	Truth functionally complete connectives	162	Quantified Boolean formulas (QFB)
163	First order theory of equality	164	Modal logic S5-Satisfiability
165	Modal logic provability	166	Predicate logic without negation
167	Conjunctive satisfiability with functions and inequalities	168	Minimum axiom set
169	First order subsumption	170	Second order instantiation
171	Finite state automaton inequivalence	172	Two-way finite state automaton non-emptiness
173	Linear bounded automaton acceptance	174	Quasi-real-time automaton acceptance
175	Non-erasing stack automaton acceptance	176	Finite state automata intersection
177	Reduction of incompletely specified automata	179	Minimum inferred finite state automaton
179	Regular expression inequivalence	180	Minimum inferred regular expression
181	Reynolds covering for context-free grammars	182	Covering for linear grammars
183	Structural inequivalence for linear grammars	184	Regular grammar inequivalence
185	Non-lr (k) context-free grammar	186	ETOL grammar non-emptiness
187	Context-free programmed language membership	188	Quasi-real-time language membership
189	ETOL language membership	190	Context-sensitive language membership
191	Tree transducer language membership	192	Register sufficiency

By examining the transformations digraph of *NP*-complete problems, we found out that transformations are grouped like grape clusters, which suggests the definition of a family of polynomial transformations.

Definition of family of polynomial transformations: the set of *NP*-complete problems $P = [p_1, p_2, \ldots, p_n]$ that consists of one parent problem and several child problems, such that each child problem is related to the parent problem trough a polynomial or pseudo-polynomial transformation. For example, the polynomial transformation family of 2-Partition Problem (or Partition Problem) consists of the following transformations (Fig. 7): Partition $\leq_P$ Minimum Sum of Squares, Partition $\leq_P$ Bin Packing, Partition $\leq_P$ Kth Largest m-tuple, Partition $\leq_P$ Expected Retrieval Cost, Partition $\leq_P$ Shortest Weight Constrained Path, Partition $\leq_P$ Multiprocessor Scheduling, Partition $\leq_P$ Sequencing to minimize tardy task weight, Partition $\leq_P$ Sequencing with deadlines and set-up times, Partition $\leq_P$ Randomization test for matched pairs, Partition $\leq_P$ Shapley–Shubik voting power, Partition $\leq_P$ Scheduling to minimize weighted completion time, Partition $\leq_P$ Open-shop scheduling, Partition $\leq_P$ Production planning, Partition $\leq_P$ Quadratic programming, Partition $\leq_P$ Continuous multiple choice knapsack, Partition $\leq_P$ Cosine product integration, Partition $\leq_P$ Max Cut, Partition $\leq_P \geq$ Knapsack, Partition $\leq_P \geq$ Subset Sum.

4. Differences between polynomial reductions and polynomial transformations

The polynomial reduction approach mentions that a language L_1 of a decision problem is reducible in polynomial time to a language L_2 of a decision problem ($L_1 \propto L_2$) if there exists a function f computable in polynomial time, for all $x \in L_1$ if and

Table 3
List of problems (part 3).

Number	Name of problem	Number	Name of problem
193	Feasible register assignment	194	Register sufficiency for loops
195	Code generation on a one-register machine	196	Code generation with unlimited registers
197	Code generations for parallel assignments	198	Code generation with address expressions
199	Code generation with unfixed variable locations	200	Ensemble computation
201	Microcode bit optimization	202	Inequivalence of programs with arrays
203	Inequivalence of programs with assignments	204	Inequivalence of finite memory programs
205	Inequivalence of loop programs without nesting	206	Inequivalence of simple functions
207	Strong inequivalence of Ianov schemes	208	Strong equivalence for monadic recursion schemes
209	Non-containment for free B-Schemes	210	Non-freedom for loop-free program schemes
211	Programs with formally recursive procedures	212	Betweenness
213	Cyclic ordering	214	Non-liveness of free choice Petri nets
215	Reachability for 1-conservative Petri nets	216	Finite function generation
217	Permutation generation	218	Decoding of linear codes
219	Shapley–Shubik voting power	220	Clustering
221	Randomization test for matched pairs	222	Maximum likelihood ranking
223	Matrix domination	224	Matrix cover
225	Simply deviated disjunction	226	Decision tree
227	Minimum weight and/or graph solution	228	Fault detection in logic circuits
229	Fault detection in directed graphs	230	Fault detection with test points
231	Graph isomorphism	232	Subgraphs homeomorphism (for fixed graph H)
233	Graph Genus	234	Chordal graph completion
235	Chromatic index	236	Spanning tree parity problem
237	Partial order dimension	238	Precedence constrained 3-processor scheduling
239	Linear programming	240	Total unimodularity
241	Composite number	242	Minimum length triangulation
243	Generalized hex	244	Generalized geography
245	Generalized Kayles	246	Sequential truth assignment
247	Variable partition truth assignment	248	Sift
249	Alternating hitting set	250	Alternating maximum weighted matching
251	Annihilation	252	$N \times N$ checkers
253	$N \times N$ Go	254	Left–right Hackenbush for redwood furniture
255	Square-tiling	256	Crossword puzzle construction
257	Generalized instant insanity	258	Partition into clique (clique cover)
259	Covering by cliques	260	Covering by complete bipartite subgraphs
261	Set covering	262	String to string correction
263	Grouping by swapping	264	Max Cut
265	Minimum cut into bounded sets	266	Set splitting (hypergraph 2-colorability)
267	Directed Hamiltonian path	268	Steiner tree
269	Job sequencing	270	Maximum leaf spanning tree
271	Safety of file protection systems	272	Comparative Vector Inequalities
273	Network Reliability	274	Sequencing to minimize maximum cumulative cost

Table 4
List of problems (part 4).

Number	Name of problem	Number	Name of problem
275	2-processor flow-shop with bounded buffer	276	Simultaneous divisibility of linear polynomials
277	Regular expression non-universality	278	Optimal Linear arrangement
279	Directed optimal linear arrangement	280	Consecutive ones matrix argumentation
281	Sequencing to minimize weighted completion time	282	Crossing number
283	Rooted tree arrangement	284	Rooted tree storage assignment
285	Partition into forests	286	Sparse matrix compression
287	Conjunctive query foldability	288	Two-dimensional consecutive sets
289	Perl regular expression	290	Graph k-colorability (chromatic number)
291	Graph 3-colorability	292	Graph 4-colorability
293	Graph 5-colorability	294	Graph motion planning
295	Travel robot localization	296	Circuit SAT

only if $f(x) \in L_2$ [1]. This implies reducing a problem to another one through a subroutine that solves L_2 in polynomial time. The function f for the reduction function and algorithm or reduction to algorithm F of polynomial time is said to compute f [1].

On the other hand, the polynomial transformation approach mentions that a language L_1 of a decision problem A is transformable in polynomial time to a language L_2 of a decision problem B ($L_1 \leq_P L_2$) if there exists a transformation (algorithm, compiler, graph transformation rules) f from problem A to problem B. This implies a transformation from each instance x of L_1 to an instance $f(x)$ of L_2.

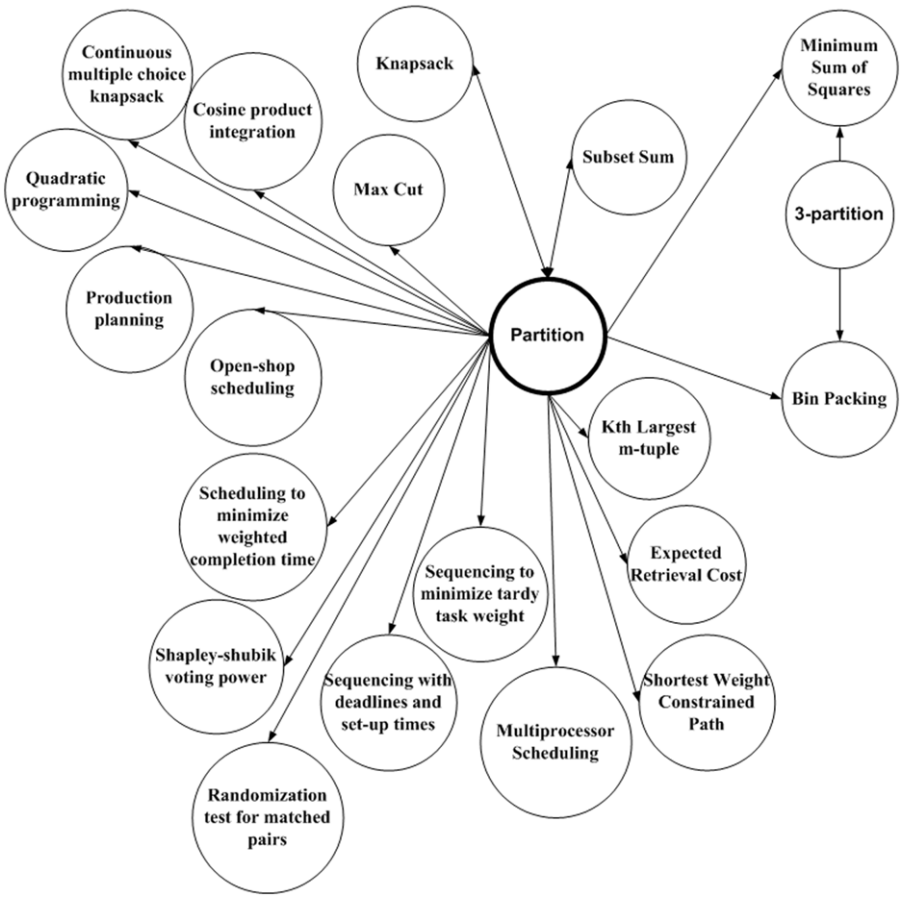

Fig. 7. Graph of the Partition transformation family of 2-partition.

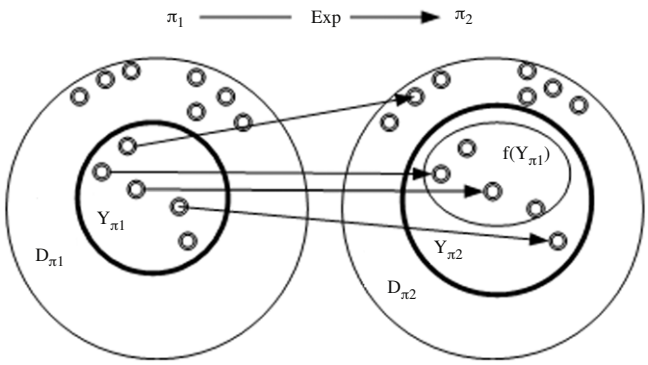

Fig. 8. Polynomial reduction.

A difference between a polynomial reduction (Fig. 8) and a polynomial transformation (Fig. 10) is that in the first one problems are transformed through reduction expressions (*exp*) from yes-instances (Y_{π_1}) of a problem (π_1) to yes-instances (Y_{π_2}) of another problem (π_2), (Fig. 9); and sometimes from yes-instances (Y_{π_1}) of problem π_1 to no-instances (D_{π_2}) of problem π_2 (see Sections 2.1.1–2.1.3).

Unlike polynomial reductions, polynomial transformations (Fig. 10) transform using f (a function, algorithm, graph theory or formal language theory) from yes-instances (Y_{π_1}) of a problem (π_1) to yes-instances (Y_{π_2}) of another problem (π_2) (see Sections 2.2.1–2.2.3).

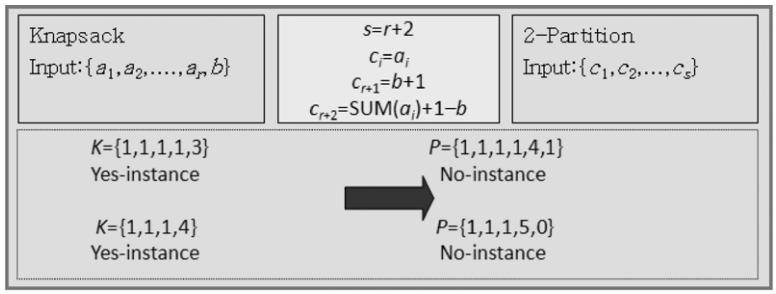

Fig. 9. Polynomial reduction from yes-instance to no-instance.

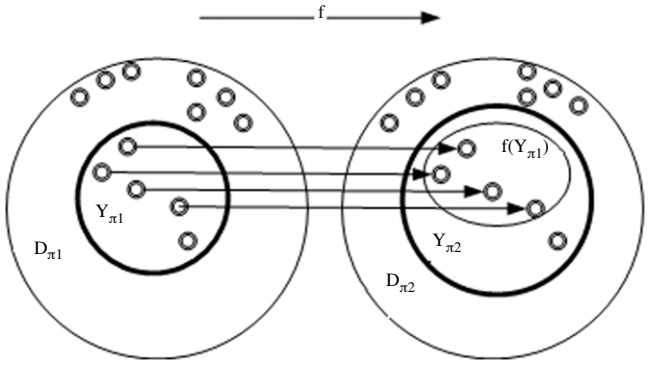

Fig. 10. Polynomial transformation.

5. Final remarks and discussion

Problem transformation is a mechanism that is useful for: finding out if a problem belongs to a class of problems, determining if a problem is more complex than another, and for helping solve complex real-life optimization problems for which no algorithms can be found that guarantee to yield exact solutions. Polynomial transformation is possible through transformation expressions, *NP*-completeness theory (transformation algorithms), graph theory and formal language theory. Each option has advantages and disadvantages. We think that, if one needs to carry out a polynomial transformation in a practical way, one should use formal language theory or *NP*-completeness theory.

The results of this investigation have uncovered a possible flaw in the following well-known definition of *NP*-completeness: "a language L is defined to be *NP*-complete if $L \in NP$ and, for all other languages $L' \in NP$, $L' \propto L$" [2]. If we consider that L' is the language that encodes problem A and L is the language for problem B, this implies that any *NP*-complete problem A can be transformed into *NP*-complete problem B; which does not always occur. For example, let us consider the Bin Packing Problem (1D-BPP), which will take the role of problem A, and the 2-Partition Problem (2-PAR), which will take the role of B. First, notice in Fig. 7 that the transformation goes from B to A, so the following question arises: is it possible to transform from A to B? The answer to this question is that, though we could devise a function that transformed A instances to B instances (and we did so in Section 2.2.3), the problem is that there exist yes-instances in A, that map into no-instances in B; for example, consider a 1D-BPP yes-instance in which the sum of all the item sizes is an odd number, then the transformed 2-PAR instance would be a no-instance; and therefore, this transformation is invalid for *NP*-completeness. From this reflection, it looks as if this issue deserves further study.

As future work, we think it could be convenient to perform polynomial transformations using formal language theory for those problems that were transformed through reduction expressions.

References

[1] S.A. Cook, The complexity of theorem-proving procedures, in: STOC'71: Proceedings 3rd Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1971, pp. 151–158, doi:10.1145/800157.805047.
[2] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of *NP*-Completeness, W.H. Freeman and Company, New York, 1979.
[3] L.R. Ford, D.R. Fulkerson, Flows in Networks, Princeton University Press, Princeton, NJ, 1962.
[4] O.B. Dantzig, W. Blattner, M.R. Rao, Finding a cycle in a graph with minimum cost to times ratio with applications to a ship routing problem, in: P. Rosenstiehl (Ed.), In Theory of Graphs, Dunod, New York, 1967, pp. 77–84.
[5] E.L. Post, Recursively enumerable sets of positive integers and their decision problems, Bull American Mathematical Society 50 (1944) 284–316. doi:10.1090/S0002-9904-1944-08111-1.
[6] R.E. Ladner, N.A. Lynch, A.L. Selman, Comparison of polynomial-time reducibilities, in: Proceedings of the Sixth Annual ACM Symposium on Theory of Computing, STOC 74, ACM, New York, 1974, pp. 110–121. doi:10.1145/800119.803891.

[7] R.E. Ladner, N.A. Lynch, A.L. Selman, A comparison of polynomial time reducibilities, Theoretical Computer Science 1 (2) (1975) 103–123. doi:10.1016/0304-3975(75)90016-X.
[8] A.R. Meyer, L.J. Stockmeyer, The equivalence problem for regular expressions with squaring requires exponential space, in: 13th Annual IEEE Symposium of Switching and Automata Theory, 1972, pp. 125–129, doi:10.1109/SWAT.1972.29.
[9] R.M. Karp, Reducibility among combinatorial problems, in: R.E. Miller, J.W. Thatcher (Eds.), Complexity of Computer Computations, Plenum Press, New York, 1972, pp. 85–103.
[10] S. Martello, P. Toth, Knapsack Problems: Algorithms and Computer Implementations, John Wiley & Sons, England, 1991, pp. 221–239.
[11] J. Nievergelt, M. Brandle, M. Dreier, GrapBench Versión 2.1, Swiss Federal Institute of Technology Zurich, November, 2005.
[12] J.A. Ruiz-Vanoye, J. Pérez-Ortega, R.A. Pazos R., O. Díaz-Parra, H.J. Fraire Huacuja, J. Frausto-Solís, G. Reyes-Salgado, L. Cruz-Reyes, Application of formal languages in the polynomial transformations of instances between NP-complete problems, Unpublished manuscript.
[13] J. Hopcroft, J. Ullman, Formal Languages and their Relation to Automata, Addison-Wesley, 1969.