Journal of Zhejiang University-SCIENCE C (Computers & Electronics)
ISSN 1869-1951 (Print); ISSN 1869-196X (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

Application of formal languages in polynomial transformations of instances between NP-complete problems

Jorge A. RUIZ-VANOYE†1, Joaquín PÉREZ-ORTEGA2, Rodolfo A. PAZOS RANGEL3, Ocotlán DÍAZ-PARRA1, Héctor J. FRAIRE-HUACUJA3, Juan FRAUSTO-SOLÍS4, Gerardo REYES-SALGADO5, Laura CRUZ-REYES3

(1DACI, Universidad Autónoma del Carmen, Cd. del Carmen 24180, Mexico)
(2Computer Science, Centro Nacional de Investigación y Desarrollo Tecnológico, Cuernavaca 62490, Mexico)
(3Systems and Computer Science, Instituto Tecnológico de Ciudad Madero, Ciudad Madero 89440, Mexico)
(4Informatics, Universidad Politécnica del Estado de Morelos, Jiutepec 62560, Mexico)
(5Computer Science, Instituto Tecnológico de Cuautla, Cuautla 62745, Mexico)
†E-mail: jorge@ruizvanoye.com

Received Dec. 3, 2012; Revision accepted Mar. 19, 2013; Crosschecked July 12, 2013

Abstract: We propose the usage of formal languages for expressing instances of NP-complete problems for their application in polynomial transformations. The proposed approach, which consists of using formal language theory for polynomial transformations, is more robust, more practical, and faster to apply to real problems than the theory of polynomial transformations. In this paper we propose a methodology for transforming instances between NP-complete problems, which differs from Garey and Johnson's. 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 is intended for extrapolating some known characteristics, phenomena, or behaviors from a problem *A* to another problem *B*. This extrapolation could be useful for predicting the performance of an algorithm for solving *B* based on its known performance for problem *A*, or for taking an algorithm that solves *A* and adapting it to solve *B*.

Key words: Formal languages, Polynomial transformations, NP-completeness
doi:10.1631/jzus.C1200349 **Document code:** A **CLC number:** TP301.5

1 Introduction

The theory of computational complexity is a subject of international interest of scientific, technological, and enterprise organizations (Ruiz-Vanoye and Díaz-Parra, 2011). The computational complexity contains diverse elements such as complexity of the classes of problems (P, NP, NP-hard and NP-complete, etc.), complexity of algorithms (it is a way to classify the efficiency of an algorithm by means of execution time, to solve a problem with the worst-case input), complexity of instances (it is a computational complexity measure to determine the complexity of the problem instances), and the complexity of other elements (Ruiz-Vanoye and Díaz-Parra, 2011), for example, the complexity of neural networks (Bao *et al.*, 2012).

In computational complexity theory, there exists a polynomial transformation from an NP-complete problem *A* to another NP-complete problem *B* (Garey and Johnson, 1979). The investigations that served as foundation for the development of complexity theory were the following:

1. The questions posed by D. Hilbert in 1929 (Cook, 1983) were key for the beginning of computational complexity: (1) Is mathematics complete, in the sense that each mathematical statement can be proved or not? (2) Is mathematics consistent, in the sense that a statement and its negation cannot be proved simultaneously? (3) Is mathematics decidable,

in the sense that there exists a defined method that can be applied to any mathematical statement and that determines if that statement is true? Hilbert's goal was to create a formal mathematical system (complete and consistent), in which all statements could be precisely formulated. His idea was to find an algorithm that determines the truth or falseness of any statement in the formal system. In German, he called this problem 'Entscheidungsproblem', meaning 'decision problem'.

2. Turing (1937) argued that Hilbert's third question (the Entscheidungsproblem) could be approached with the help of a machine, at least with the notion of an 'abstract machine'. Turing's purpose, when describing the machines that nowadays bear his name, was to reduce calculations to their most concise essential features, describing in a simple way some basic procedures.

3. Shannon (1948) proposed the complexity of Boolean circuits (or combinatorial complexity). For this measure it is convenient to assume that a function (f) transforms finite bit chains (A) into finite bit chains (B), and the complexity of f is the size of the smallest Boolean circuit that calculates f for all the inputs of size n.

4. Rabin (1959) posed a general question: What does it mean that f is more difficult to compute than g? To answer this question, he conceived an axiomatic framework (emphasizing the interrelationship between seemingly diverse problems and methods for research in the theory of complexity of computations) that was the foundation for the development of Blum's abstract complexity theory.

5. Solomonoff (1960) is the first to propose the Kolmogorov complexity (Solomonoff, 1964a; 1964b; Kolmogorov, 1965), which provides a measure for the combinatorial complexity of an individual string x.

6. Cobham (1964) emphasized the term 'intrinsic'; i.e., he was interested in an independent-machine theory. He asked whether multiplication was more complex than addition and considered that the question could not be answered until the theory was developed. Additionally, he defined the class L of functions, i.e., functions on real numbers computable in time bounded by a polynomial in the decimal length of the input.

7. Hartmanis and Stearns (1965) introduced the fundamental notion of 'complexity measure', which is defined as the computation time on a multi-tape Turing machine.

8. Stockmeyer (1979) mentioned the computational complexity of some problems and common measures of the computational resources used by an algorithm: time, the number of steps executed by the algorithm, and space (the amount of memory used by the algorithm).

9. Sipser (1983) proposed the CD-complexity. $\mathrm{CD}^t(x)$ is the size of the smallest program that distinguishes x from all other strings at time t. Wozniakowski (1985) and Traub *et al.* (1988) proposed the information-based complexity (IBC), which seeks to develop general results about the intrinsic difficulty of solving problems where available information is partial or approximate and to apply these results to specific problems.

10. Quantum circuits were first defined by Deutsch (1989) and study of quantum circuit complexity (quantum computational complexity) was initiated by Yao (1993).

11. Orponen (1990) introduced a measure for the computational complexity of individual instances of a decision problem.

Computational complexity theory is related to the measurement of the difficulty (complexity) of calculation. The complexity measurement shows how many steps are needed to evaluate any algorithm with a specific parameter (Hartmanis and Stearns, 1965; Papadimitriou, 1994). Some definitions of complexity classes are mentioned related to this investigation:

Definition 1 (P class) P class is the class of recognizable languages by a deterministic one-tape Turing machine in polynomial time, introduced by Karp (1972). Although the first individual that used an idea which could be considered equivalent to P was Edmonds (1965), Edmonds considered the algorithms in polynomial time as tractable algorithms.

Definition 2 (NP class) NP class is the class of recognizable languages by a non-deterministic one-tape Turing machine in polynomial time, introduced by Karp (1972). The NP class includes diverse practical problems from businesses and industry. Garey and Johnson (1979) presented many examples of NP problems. The first mathematician who used a term that can be taken as equivalent to NP, however, was James Bennett, who used the term 'extended positive rudimentary relations' based on logical operators instead of computing machines (Bennett, 1962). Later, Cook (1971) characterized problems using the term L^+ for the then well-known NP class defined by Karp (1972).

Definition 3 (NP-easy class) NP-easy class is the class of problems that are recognizable in polynomial time by a Turing machine with one oracle (subroutine); i.e., a problem *A* is said to be NP-easy if it can be Turing reduced to some NP-complete problem (Jonsson and Bäckström, 1994). In other words, a problem *X* is NP-easy if, and only if, there exists a problem *Y* in NP such that *X* is Turing reducible (with oracle) in polynomial time to *Y* (Jonsson and Bäckström, 1994). Informally, NP-easy problems are those problems that are at most as difficult as NP.

Definition 4 (NP-hard class) A problem *A* is NP-hard if each problem *B* in NP is reducible to *A* (Garey and Johnson, 1979; Papadimitriou and Steiglitz, 1982). Informally, it is the class of problems classified as problems of combinatorial optimization at least as complex as NP.

Definition 5 (NP-equivalent class) NP-equivalent class is the class of problems that are considered to be both NP-easy and NP-hard (Jonsson and Bäckström, 1994).

Definition 6 (NP-complete class) A language *L* is NP-complete if *L* is in NP, and satisfiability$\leq_P L$ (Cook, 1971; Karp, 1972). NP-complete class is the class of problems classified as decision problems. A debate among researchers exists on whether Cook defined the NP-complete term or in fact he defined *L* (polynomial) complete, although, generally, the definition of this term is attributed to him. Cook (1971) hinted at the NP-complete notion and demonstrated that the 3-satisfiability problem and the sub-graph problem are NP-complete. Karp (1972) presented 21 polynomial reductions to NP-complete problems. Informally, NP-complete problems are those that are complete in the NP class, i.e., the most difficult to solve in NP. Bennett *et al.* (1994) showed that any quantum algorithm should solve a basic NP-complete problem (3-SAT) efficiently.

In this paper, we propose the usage of formal languages to express instances of NP-complete problems for their application in polynomial transformations. The proposed approach of using formal language theory for a polynomial is more robust and practical from a computational point of view to apply to real problems than the theory of polynomial transformations, because it is not necessary to determine equations of polynomial transformation that encode a transformation algorithm exactly at yes-instances into yes-instances. It needs only to know the source language, the target language, and the hard restrictions of the problem to apply tools of formal languages, some of which are automated for exploiting the full characteristics of the languages.

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 is intended for extrapolating some known characteristics, phenomena, or behaviors from a problem *A* to another problem *B*.

2 Related works

This investigation is related to the topics of formal languages and polynomial transformations. There exist several definitions of formal languages. The following are the most relevant:

Brown (1960) described engineering languages with the objective of making a language different from natural language. The engineering languages are languages that are designed to specified objective criteria, and modeled to meet the criteria.

Hopcroft and Ullman (1969) defined language as any set V^* of sentences on an alphabet *V*. A sentence of an alphabet is any string of finite length composed of symbols from alphabet *V*. For example, if $V=\{0, 1\}$, then $V^*=\{\text{empty}, 0, 1, 00, 01, 10, 11, 000, ...\}$. An alphabet is any finite set of symbols (digits, Latin and Greek letters in lower case and upper case, symbols #, and others).

Cook (1971) defined a language as a set *G* of chains of symbols on a fixed, large, and finite alphabet $\{0, 1, *\}$.

Karp (1972) defined a language as a subset of Σ^* (the set of all the finite chains of 0's and 1's). Karp defined NP-complete problems (*L*-complete) as follows: call *L* (polynomial) complete if $L\in\text{NP}$ and every language in NP is reducible to *L*; i.e., it is the class of recognizable languages in polynomial time by a non-deterministic one-tape Turing machine.

Garey and Johnson (1979) mentioned a language in the following way: for any finite set Σ of symbols, denote by Σ^* the set of all finite strings of symbols from Σ, if $\Sigma=\{0, 1\}$. For example, Σ^* consists of the empty string, the strings 0, 1, 00, 01, 10, 11, 000, 001, and all other finite strings of 0's and 1's. If L is a subset of Σ^*, we say that L is a language over the Σ alphabet.

For polynomial transformations several definitions exist (Ruiz-Vanoye *et al.*, 2011):

Karp (1972) defined a polynomial reduction as follows: considering two languages L and M, then L is reducible to M if there exists a function $f\in M\Leftrightarrow x\in L$.

Cook (1971) defined a polynomial reduction as follows: 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 of M with input w halts in $Q(|w|)$ steps ($|w|$ is the length of w) and ends in the accept state iff $w\in S$.

Garey and Johnson (1979) defined a polynomial transformation from a language $L_1\subseteq\Sigma_1^*$ to a language $L_2\subseteq\Sigma_2^*$ as a function f: $\Sigma_1^*\rightarrow\Sigma_2^*$ that satisfies the following two conditions: (1) There is a polynomial time deterministic Turing machine (DTM) program that computes f. (2) For all $x\in\Sigma_1^*$, $x\in L_1$ if and only if $f(x)\in L_2$.

In this study a new approach for instance transformation between NP-complete problems is proposed using formal language theory. The related works relevant to this research are shown in Table 1.

Table 1 Related works

Research	C1	C2	C3	C4
Karp (1972)	Y	N	N	N
Cook (1971)	Y	N	N	N
Garey and Johnson (1979)	Y	Y	N	N
This work	Y	Y	Y	Y

C1: polynomial reduction between NP-complete problems; C2: transformation of yes-instance to yes-instance; C3: transformation using formal languages of NP-complete problems; C4: transformation and solution of NP-complete problems. Y: yes; N: no

3 Methodology of polynomial transformation of instances between NP-complete problems using formal languages

We propose the usage of the theory of formal languages (compiler techniques) in polynomial transformations. A compiler is a computer program (or a set of programs) that translates text written in a source language into another target language (Aho *et al.*, 1986). For this work the phases of a compiler are:

Lexical analysis: recognize and convert the character stream from the input source program or sequence of characters to valid words of the language or tokens.

Syntactic analysis: consider the sequence of tokens for possible valid constructs of the language.

Semantic analysis: determine the meaning of the language.

Error handling: detect the lexical, syntactic, semantic, and logical errors.

Language generation: generate the target code or the target language.

The proposed notion for polynomial transformations is similar to the translation from one language to another, for example, from the English language to the Spanish language; instead of making a translation between languages, the translation is carried out between instances of NP-complete problems. In this approach, formal languages are generated for defining instances of NP-complete problems, to use them in a polynomial transformation (using a compiler) from a formal language L_1 (corresponding to an NP-complete problem) to a formal language L_2 (corresponding to another NP-complete problem).

A language corresponds to an NP-complete problem when the values of the instance parameters of the problem can be codified in a hypothetical language (Ruiz-Vanoye *et al.*, 2010). For example, the parameters of the instance of the one-dimensional bin-packing problem (1D-BPP) are: num=overall number of items, w_i=size of item i, c=bin capacity, k=number of bins. Therefore, the hypothetical language of 1D-BPP is L_1={BPP={num; w_1, w_2, w_3, w_4, w_5, w_6; c; k}}, and the hypothetical language with the values of the instance is L_1={BPP={6; 10, 20, 20, 10, 5, 5; 20; 4}}. The hypothetical language has an alphabet Σ and a Backus-Naur form (BNF) (Backus, 1959) grammar or grammatical rules for the 1D-BPP.

Fig. 1 shows the methodology proposed for transforming instances from an NP-complete problem to instances of another NP-complete problem using formal language theory.

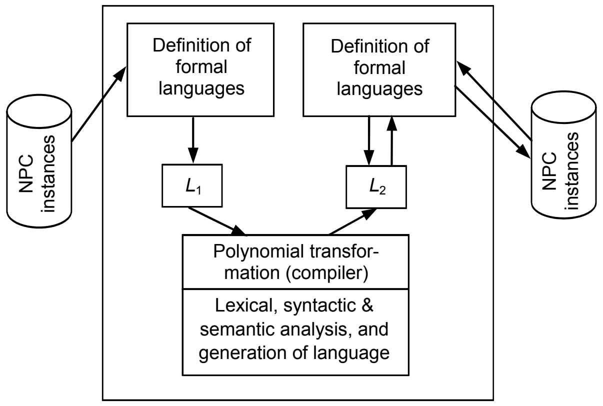

Definition of formal languages
Definition of formal languages
NPC instances
NPC instances
L_1
L_2
Polynomial transformation (compiler)
Lexical, syntactic & semantic analysis, and generation of language

Fig. 1 Polynomial transformation using formal language theory

The steps for polynomial transformation using formal language theory are:

1. Select an NP-complete problem *A* (source problem).

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 the NP-complete problem *B*.

5. Construct a compiler that transforms in polynomial time ($L_1 \leq_P L_2$) a source language L_1 into a target language L_2.

6. Optionally, it is possible to add to the phase of language generation an algorithm that solves target language L_2.

Our proposal (Fig. 2) is a new codification scheme of NP-complete problems based on formal languages, i.e., a new way to transform instances using a compiler from source language L_1 (which defines yes-instances of the NP-complete problem or feasible solutions Y_{π_1}) to target language L_2 (which defines yes-instances of the NP-complete problem Y_{π_2}). The *D* defines no-instances of both NP-complete problems.

In the phase of lexical analysis, source language L_1 is converted into tokens. In the phase of syntactic analysis, the tokens are grouped into grammatical sentences.

In the phase of semantic analysis, semantic errors are detected in the source language, and the information is stored for use in the phase of language generation. Semantic error detection involves checking the restrictions to obtain a formal language L_2.

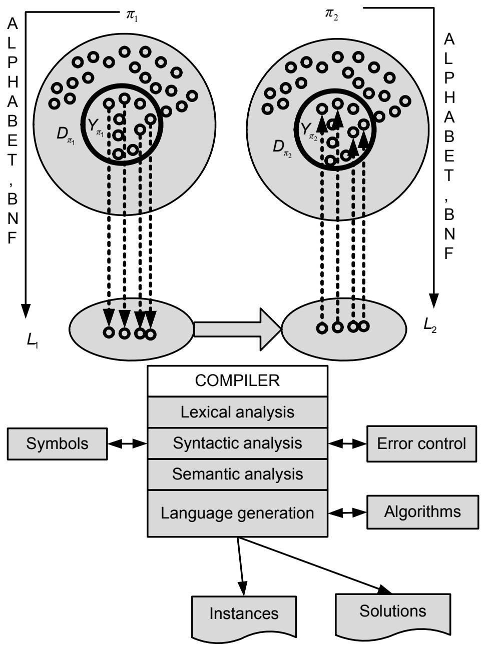


Fig. 2 Methodology of polynomial transformation using formal language theory

In the language generation phase, we obtain the target language (the yes-instance) which uses the section of algorithms (procedures that realize the execution of diverse algorithms) for finding solutions to the NP-complete instances of the L_2.

The section of error control detects the lexical, syntactic, semantic, and logical errors of the transformations from L_1 to L_2.

Fig. 3 shows the methodology of polynomial transformation of Garey and Johnson (1979). The NP-complete approach consists of four steps for reducing NP-complete problems (Garey and Johnson, 1979):

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.

In Table 2 there exist some differences between Garey and Johnson (1979)'s approach and ours. The main difference is that, our approach includes (among other things) checking the hard restrictions of the NP-complete problems by the phases of a compiler, for extrapolating some known characteristics, phenomena, or behaviors from a problem *A* to another problem *B*.

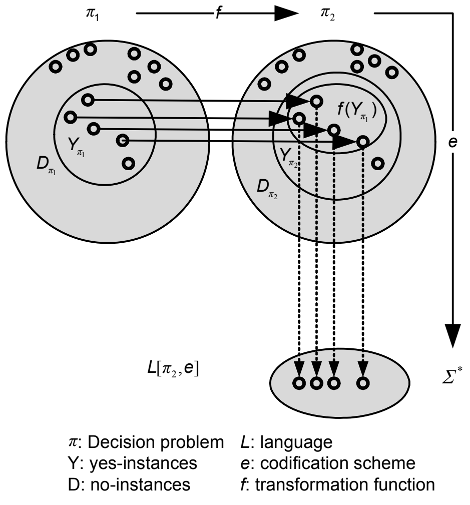


Fig. 3 Methodology of polynomial transformation of Garey and Johnson (1979)

Table 2 Differences between the two approaches

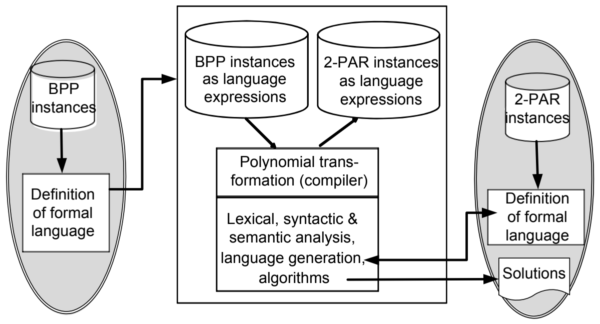
Characteristic	Garey and Johnson (1979)'s	Ours
Transformation between NP-complete problems	Y	Y
Transformation of yes-instance to yes-instance	Y	Y
Transformation algorithm	Y	N
Compiler for the transformation	N	Y
Lexical verification in the transformation	Y	Y
Syntactic verification in the transformation	Y	Y
Semantic verification in the transformation	N	Y
Solution of the transformed instance	N	Y
Definition of a formal language for the source problem (L_1) by means of a codification scheme	N	Y
Definition of a formal language for the target problem (L_2) by means of a codification scheme	Y	Y
Administration of errors during the transformation	N	Y
Robust, practical, and fast development	N	Y

Y: yes; N: no

In the semantic verification of the polynomial transformation, the compiler verifies the hard restriction to validate the feasible instance (yes-instance), the solution of the transformed instance, and the administration of errors during the transformation by a computational tool (robust and practical). This tool is robust because the full characteristics of the languages are exploited, and practical from a computational point of view, because an automated tool can do the polynomial transformation and it needs only to know the source language, the target language, and the hard restrictions of the problem.

4 Experimentation

For the experimentation the following software was used: Microsoft Visual C^{++} ver. 6.0 for coding the instances generator and the solution algorithms, and Parser Generator ver. 2.07 from Bumble-Bee Software Ltd. (which comprises Lex/Flex and Yacc/Bison) for the compiler that transforms instances of the NP-complete problems. Fig. 4 shows the experimentation scheme of the polynomial transformation using formal language.



BPP instances
Definition of formal language
BPP instances as language expressions
2-PAR instances as language expressions
Polynomial transformation (compiler)
Lexical, syntactic & semantic analysis, language generation, algorithms
2-PAR instances
Definition of formal language
Solutions

Fig. 4 Process for polynomial transformation

The experiments were carried out using the following NP-complete problems: the BPP and the 2-partition problem (2-PAR).

Definition of 2-PAR (Garey and Johnson, 1979; Martello and Toth, 1991) is: 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.

$$U=\{\boldsymbol{u}_1,\boldsymbol{u}_2,\ldots,\boldsymbol{u}_n\}, \tag{1}$$

$$A\subset U, \tag{2}$$

$$A^c=U-A, \tag{3}$$

$$A\cap A^c=\varnothing, \tag{4}$$

$$\sum_{i\in A}\boldsymbol{u}_i=\sum_{j\in A^c}\boldsymbol{u}_j. \tag{5}$$

The input parameters of 2-PAR are shown in Table 3, and the output parameters in Tables 4 and 5.

Table 3 Input parameters for the 2-PAR problem

Input	Meaning
n	Overall number of integers
$\boldsymbol{u}_i$	Vector of positive integer numbers (i=1, 2, …, n)

Table 4 Output parameters of the 2-PAR decision problem

Output	Meaning
True/False	Positive or negative answer to the question

Table 5 Output parameters of the 2-PAR combinatorial optimization problem

Output	Meaning
A	Subset of integer numbers of the solution
A^c	Complement subset of integer numbers of the solution

Definition of 1D-BPP (Martello and Toth, 1991) is: given a finite set U of n objects with weights w_1, w_2, …, w_n, a positive integer number c that represents the bin capacity, and a positive integer number K (maximum number of bins), the problem consists of determining if there exists a partition of U consisting of disjoint sets U_1, U_2, …, U_K such that the sum of the object weights in each U_i is c or less. The output parameters of 1D-BPP are shown in Table 6, and the output parameters in Tables 7 and 8.

Table 6 Input parameters of 1D-BPP

Input	Meaning
n	Number of objects
w_i	Object weight (size) (i=1, 2, ..., n)
c	Capacity of bins
K	Maximum number of bins

Table 7 Output parameters of the 1D-BPP decision problem

Output	Meaning
True/False	Positive or negative answer to the question

Table 8 Output parameters of the 1D-BPP combinatorial optimization problem

Output	Meaning
M	Value of the optimal solution (smallest number of bins)
A_i	Subset of objects assigned to each bin in the solution (i=1, 2, …, n)

We generated a sample of 1D-BPP instances. To obtain a confidence level of 99% for the sample, 27 strata were defined, and 82 instances were generated for each stratum, which yielded an overall of 2214 1D-BPP instances being used in the polynomial transformation. For the generated instances, we used the nomenclature: e (stands for strata), # (number of strata), i (instance), # (number of instances), .bpp (extension of the file) (e.g., e1i21.bpp). The ranges of the characteristics of the generated instances are shown in Table 9.

Table 9 Ranges of characteristic values

Characteristic	Range
Number of objects, n	S (10−333), M (340−660), B (670−1000)
Object size, $\bar{s}$	S (10−8325), M (40−16 500), B (670−25 000)
Bin capacity, c	S (50−1 386 112), M (6800−5 445 000), B (224 450−12 500 000)

S=small, M=medium, B=big

Hereupon, the process of polynomial transformation from 1D-BPP to 2-PAR (1D-BPP$\leq_P$2-PAR) is described, which uses formal language theory:

Step 1: Select an NP-complete problem A (source problem) for polynomial transformation. 1D-BPP was selected since it is a problem widely known to be NP-complete (Martello and Toth, 1991).

Step 2: Define a formal language L_1 (source language) for the NP-complete problem A. For example, for the 1D-BPP we defined an alphabet Σ={0, 1, 2, 3, 4, 5, 6, 7, 8, 9, '{', '}', ',', ';', '=', 'B', 'P'}, and the following BNF grammar:

```
<instance>:=<NameProblem> <Equal> <sentences>;
<sentences>:=<KOpen> <TNum> <semicolon>
        <Num> <semicolon> <NumCap> <KClose>;
<TNum>:=<Num>;
<NumCap>:=<Num>;
<Num>:=<Num> <Comma> | <Integer>;
<Integer>:=<Digit>{<Digit>}*;
<Digit>:=0|1|2|3|4|5|6|7|8|9;
<NameProblem>:='BPP';
<Equal>:='=';
<KOpen>:='{';
<KClose>:='}';
<semicolon>:=';';
<Comma>:=',';
```

Step 3: Select an NP-complete problem B (target problem). We selected 2-PAR, since it is known to be NP-complete (Martello and Toth, 1991).

Step 4: Define a formal language L_2 (target language) for the NP-complete problem B. For example, for 2-PAR, we defined an alphabet Σ={0, 1, 2, 3, 4, 5, 6, 7, 8, 9, '{', '}', ',', ';', '=', 'A', 'P', 'R'}, and a BNF grammar:

```
<instance>:=<NameProblem> <Equal> <sentences>;
<sentences>:=<KOpen> <TNum> <semicolon>
                <Num> <KClose>;
<TNum>:=<Num>;
<Num>:=<Num> <Comma> | <Integer>;
<Integer>:=<Digit>{<Digit>}*;
<Digit>:=0|1|2|3|4|5|6|7|8|9;
<NameProblem>:='PAR';
<Equal>:='=';
<KOpen>:='{';
<KClose>:='}';
<semicolon>:=';';
<Comma>:=',';
```

Step 5: Construct a compiler that transforms from source language L_1 to target language L_2 being used to perform the polynomial transformation from 1D-BPP to 2-PAR (1D-BPP$\leq_P$2-PAR).

In the phase of lexical analysis of the compiler, a source language L_1 is transformed into tokens (from a symbol table). Fig. 5 shows the token declaration for the GNU Flex software used in the lexical phase.

```
// Declarations
A [aA]   B [bB]    C [cC]   D [dD]   E [eE]   F [fF]
G [gG]   H [hH]   I [iI]   J [jJ]   K [kK]   L [lL]
M [mM]   N [nN]   O [oO] P [pP]   Q [qQ]
R [rR]   S [sS]    T [tT]    U [uU]  V[vV]
W [wW] X [xX] Y [yY]  Z [zZ]  Digit [0-9]
// Rules
'='          return(EQUAL);
','          return(COMMA);
';'          return(SEMICOLON);
'{'          return(KOPEN);
'}'          return(KCLOSE);
[Digit]+    return(DIGSEQ);
.            { /* Ignore bad characters */ }
```

Fig. 5 Tokens for the lexical phase

In the syntactic analysis phase, tokens from source language L_1 are grouped into grammatical phrases. In the semantic analysis phase, semantic errors are detected (error handling) in source language L_1, and information resulting from these analyses is stored for the language generation phase. In the phase of error handling or error control, the lexical, syntactic, semantic, and logical errors of the transformations from L_1 to L_2 are detected. The error types shown in Table 10 were defined.

Table 10 Error types defined for the error control phase

Error	Description
101	Lexical error, symbol, or character (token) not admitted in the instance language
102	Syntactic error, ';' was expected
103	Syntactic error, ',' was expected
104	Syntactic error, INTEGERNUM was expected
105	Syntactic error, '{' was expected
106	Syntactic error, '}' was expected
107	Syntactic error, '=' was expected
108	Syntactic error, 'BPP' was expected
109	Semantic error, the instance cannot be transformed, and it does not satisfy the hard restriction of 2-PAR
110	Language generation error, impossible to create the target file
111	Language generation error, source file INSTACELIST cannot be found
112	Language generation error, impossible to create an indicator file

In the phase of semantic analysis the restrictions were checked that allow a formal language L_2 (instance) to be obtained for 2-PAR. For the phase of semantic analysis one of the hard restrictions of 2-PAR was defined in the parser generator software called GNU Bison (Levine, 2009), which states that the sum of the integer numbers (sumintegers) must be divisible by two; if not, the result of the transformation is an invalid instance for 2-PAR and the transformation is not carried out.

```
// Semantic restriction for 2-PAR
divisible=sumintegers%2;
if (divisible==0)
{ transform_instances(); }
else
{ printf("Semantic error, transformation not
realized"); }
```

The phase of language generation (instance solution) includes a process to generate language expressions for 2-PAR instances; in addition, it contains a procedure that realizes the execution of diverse algorithms for finding solutions to 2-PAR instances. The algorithms used to solve transformed instances of 2-PAR are:

1. First fit decreasing algorithm (FFD). With this algorithm the numbers are first placed in a list sorted in non-increasing order. Then each number is picked orderly from the list and placed into the first set that has enough unused space to hold it (Note: the available space of groups A and A^c is set to the sum of all the numbers that can be divided by two).

2. Best fit decreasing algorithm (BFD). The only difference between FFD and BFD is that the numbers are not placed in the first group that can hold them, but in the group with the smallest unused space that can hold them.

3. Match to first fit algorithm (MFF). It is a variation of FFD that includes complementary groups (besides A and A^c). The algorithm asks for a percentage value (which is the amount of group space that can be left empty and qualify as a 'good fit') and a number of complementary groups. Each of these complementary groups is intended for temporarily holding numbers in a unique range of values. As the list of numbers is processed, each number is examined to determine if it can be assigned to a new group with numbers of a complementary group according to the same value range and getting a good fit, or packed in a partially filled group, or packed alone in a complementary group. Finally, all the numbers in the complementary groups are extracted and packed in ordinary groups (A and A^c) according to the basic algorithm FFD, but without using number ordering.

4. Match to best fit algorithm (MBF). It is a variation of BFD and similar to MFF, except for the basic algorithm used. It uses BFD without ordering the numbers.

Incidentally, these algorithms are usually applied to solve 1D-BPP instances, so they were adopted to solve the transformed 2-PAR instances.

The results of the experimentation on the sample instances (transformed from 1D-BPP) using the preceding algorithms (30 runs were executed on each instance) are shown in Table 11.

Table 11 Results of experimentation on the instances

Instance	Solution	Time (min)*			
		FFD	BFD	MFF	MBF
e1i1	25	0	0	0	0
e1i2	225	0	0	0.01	0
e1i3	85	0	0	0	0
e1i4	225	0.01	0	0	0
e1i5	25	0	0	0	0
e1i6	105	0.01	0	0	0
e1i7	115	0	0	0	0
…	…	…	…	…	…
e27i82	3672235	0.01	0	0	0

* The value was so small (between 0 and 0.01 min) that a 0 value was reported

The results obtained from the experiments were: 1161 yes-instances and 1053 no-instances of 2-PAR transformed from a total of 2214 1D-BPP yes-instances, as well as the solution to the 1161 transformed 2-PAR instances.

By way of example, we carried out the verification of the results for one instance. Consider source language L_1 that represents a 1D-BPP yes-instance A and target language L_2 that represents a 2-PAR yes-instance B.

L_1=BPP={50; 26502, 19171, 15726, 11480, 29360, 26964, 24466, 5707, 28147, 23283, 16829, 9963, 493, 2997, 11944, 4829, 5438, 32393, 14606, 3904, 155, 294, 12384, 17423, 18718, 26502, 19171, 15726, 11480, 29360, 26964, 24466, 5707, 28147, 23283, 16829, 9963, 493, 2997, 11944, 4829, 5438, 32393, 14606, 3904, 155, 294, 12384, 17423, 18718; 181588; 4}.

In order for the compiler to be able to transform this instance (L_1), it checks (among other lexical, syntactic, and semantic aspects) a hard restriction, which states that the sum of all the object weights must be divisible by two.

sum=(26502+19171+…+17423+18718)=726352.
divisible=726352/2=363176.

The result 'divisible' indicates that the hard condition is satisfied; therefore, it is possible to transform L_1 (1D-BPP yes-instance) into L_2 (2-PAR yes-instance).

Afterwards, it is necessary, using the compiler, to transform language L_1 into language L_2. As a result of compiler execution, parameter n=50 in L_1 was transformed into n=50 in L_2. Each parameter s_i in L_1 was transformed into s_i in L_2 (Note: the transformation simply assigns the value of s_i in L_1 to s_i in L_2).

s_i de L_1: 26502, 19171, …, 17423, 18718.
s_i de L_2: 26502, 19171, …, 17423, 18718.

Parameter c=181 588 in L_1 was transformed into c=363 176 in L_2:

$$L_2=\text{PAR}=\{50;\ 26502,\ 19171,\ \ldots,\ 17423,\ 18718;\ 363176\}.$$

Finally, parameter K=4 in L_1 was transformed into two subsets of a partition.

5 Conclusions

The proposed polynomial transformation using formal language theory is similar to a translation from one language to another, so if language rules are satisfied it is possible to carry out a translation that is correct and intelligible. In this paper we have shown that the use of formal language theory makes it possible to transform a yes-instance of an NP-complete problem to a yes-instance of another NP-complete problem.

Insightful readers might notice that the example transformation from 1D-BPP to 2-PAR is carried out inversely as it is usually performed. The transformation from 2-PAR to 1D-BPP is usually realized to prove that 1D-BPP is NP-complete, assuming that 2-PAR is NP-complete. However, in our example we 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.

As a result of this investigation, we found several differences between what was mentioned in Garey and Johnson (1979) about the polynomial transformations that Karp and Cook realized, and the definitions of polynomial reductions in Cook (1971) and Karp (1972).

For future work, we think it could be convenient to use the methodology of polynomial transformation using formal language theory to transform the languages of P problems to languages of NP-complete problems.

In addition, we propose to realize investigations on families of polynomial transformations with a set of NP-complete problems (Ruiz-Vanoye *et al.*, 2011) by the methodology of polynomial transformation using formal language theory.

References

Aho, A.V., Sethi, R., Ullman, J.D., 1986. Compilers: Principles. Techniques, and Tools. Addison-Wesley, USA, p.14-16.

Backus, J.W., 1959. The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich Association for Computing Machinery (ACM) and the Association for Applied Mathematics and Mechanics (GAMM) Conference. Proc. Int. Conf. on Information Processing, p.125-132.

Bao, J., Zhou, L.J., Yan, Y., 2012. Analysis on complexity of neural networks using integer weights. *Appl. Math. Inf. Sci.*, **6**:317-323.

Bennett, J.H., 1962. On Spectra. PhD Thesis, Princeton University, USA.

Bennett, C.H., Brassard, G., Jozsa, R., Mayers, D., Peres, A., Schumacher, B., Wootters, W.K., 1994. Reduction of quantum entropy by reversible extraction of classical information. *J. Mod. Opt.*, **41**(12):2307-2314. [doi:10.1080/09500349414552161]

Brown, J.C., 1960. Loglan. *Sci. Am.*, **202**:53-63. [doi:10.1038/scientificamerican0660-53]

Cobham, A., 1964. The Intrinsic Computational Difficulty of Functions. Proc. Congress for Logic, Mathematics, and Philosophy of Science, p.24-30.

Cook, S.A., 1971. The Complexity of Theorem Proving Procedures. Proc. 3rd ACM Symp. on Theory of Computing, p.151-158.

Cook, S.A., 1983. An overview of computational complexity. *Commun. ACM*, **26**(6):400-408. [doi:10.1145/358141.358144]

Deutsch, D., 1989. Quantum computational networks. *Proc. R. Soc. Lond. A*, **425**(1868):73-90. [doi:10.1098/rspa.1989.0099]

Edmonds, J., 1965. Paths, trees, and flowers. *Canad. J. Math.*, **17**:449-467. [doi:10.4153/CJM-1965-045-4]

Garey, M.R., Johnson, D.S., 1979. Computers and Intractability: a Guide to the Theory of NP-Completeness. W.H. Freeman and Company, New York, p.1-10.

Hartmanis, J., Stearns, R.E., 1965. On the computational complexity of algorithms. *Trans. Am. Math. Soc.*, **117**(5):285-306. [doi:10.1090/S0002-9947-1965-0170805-7]

Hopcroft, J., Ullman, J., 1969. Formal Languages and Their Relation to Automata. Addison-Wesley, USA, p.1-7.

Jonsson, P., Bäckström, C., 1994. Complexity Results for State-Variable Planning under Mixed Syntactical and Structural Restriction. Proc. 6th Int. Conf. on Artificial Intelligence: Methodology, Systems, Applications, p.205-213.

Karp, R.M., 1972. Reducibility among Combinatorial Problems. *In*: Miller, R.E., Thatcher, J.W. (Eds.), Complexity of Computer Computations. Plenum Press, New York, p.85-104. [doi:10.1007/978-1-4684-2001-2_9]

Kolmogorov, A.N., 1965. Three approaches to the quantitative definition of information. *Prob. Inf. Transm.*, **1**:1-7.

Levine, J., 2009. Flex & Bison. O'Reilly Media, USA.

Martello, S., Toth, P., 1991. Knapsack Problems: Algorithms and Computer Implementations. John Wiley & Sons, England, p.221-236.

Orponen, P., 1990. On the Instance Complexity of NP-Hard Problems. Proc. 5th Annual Structure in Complexity Theory Conf., p.20-27. [doi:10.1109/SCT.1990.113951]

Papadimitriou, C., Steiglitz, K., 1982. Combinatorial Optimization: Algorithms and Complexity. Prentice-Hall, New Jersey, p.342-357.

Papadimitriou, C.H., 1994. Computational Complexity. Addison-Wesley, UK, p.260-265.

Rabin, M.O., 1959. Speed of Computation and Classification of Recursive Sets. Third Convention of Scientific Societies, p.1-2.

Ruiz-Vanoye, J.A., Díaz-Parra, O., 2011. An overview of the theory of instances computational complexity. *Int. J. Combin. Optim. Probl. Inf.*, **2**(2):21-27.

Ruiz-Vanoye, J.A., Díaz-Parra, O., Pérez-Ortega, J., Pazos, R.A., Reyes Salgado, G., González-Barbosa, J.J., 2010. Complexity of Instances for Combinatorial Optimization Problems. *In*: Al-Dahoud, A. (Ed.), Computational Intelligence & Modern Heuristics, Chapter 19. IN-TECH Education and Publishing, p.319-330. [doi:10.5772/7807]

Ruiz-Vanoye, J.A., Pérez-Ortega, J., Pazos R.A., Díaz-Parra, O., Frausto-Solís, J., Fraire-Huacuja, H.J., Cruz-Reyes, L., Martínez-Flores, J.A., 2011. Survey of polynomial transformations between NP-complete problems. *J. Comput. Appl. Math.*, **235**(16):4851-4865. [doi:10.1016/j.cam.2011.02.018]

Shannon, C.E., 1948. The mathematical theory of communication. *Bell Syst. Techn. J.*, **27**(3):379-423. [doi:10.1002/j.1538-7305.1948.tb01338.x]

Sipser, M., 1983. A Complexity Theoretic Approach to Randomness. Proc. 15th ACM Symp. on Theory of Computing, p.330-335.

Solomonoff, R., 1960. A Preliminary Report on a General Theory of Inductive Inference. Report V-131, Zator Co., Cambridge, MA.

Solomonoff, R., 1964a. A formal theory of inductive inference. Part I. *Inf. Control*, **7**(1):1-22. [doi:10.1016/S0019-9958(64)90223-2]

Solomonoff, R., 1964b. A formal theory of inductive inference. Part II. *Inf. Control*, **7**(2):224-254. [doi:10.1016/S0019-9958(64)90131-7]

Stockmeyer, L.J., 1979. Classifying the Computational Complexity of Problems. Research Report RC 7606, Mathematical Sciences Department, IBM Thomas J. Watson Research Center, Yorktown Heights, NY.

Traub, J.F., Wasilkowski, G.W., Woźniakowski, H., 1988. Information-Based Complexity. Academic Press, New York.

Turing, A.M., 1937. On computable numbers, with an application to the Entscheidungsproblem. *Proc. Lond. Math. Soc.*, **s2-42**(1):230-265. [doi:10.1112/plms/s2-42.1.230]

Wozniakowski, H., 1985. Survey of information-based complexity. *J. Compl.*, **1**(1):11-44. [doi:10.1016/0885-064X(85)90020-2]

Yao, A.C., 1993. Quantum Circuit Complexity. Proc. 34th Annual IEEE Symp. on Foundations of Computer Science, p.352-361.