International Journal of Computer Systems (ISSN: 2394-1065), Volume 03– Issue 03, March, 2016
Available at http://www.ijcsonline.com/

Solving the Pixel Puzzle under Answer Set Programming

Omar EL Khatib[A]

[A]Computer Science, Taif University, Taif, Saudi Arabia

Abstract

In this work, we present a representation and an automatic solving of a pixel puzzle using answer set programming. The puzzle has been proven previously to be NP-complete. Pixel puzzle consists of blank rectangular grid of any size with clues on the left and top of the grid. The rectangular grid is subdivided into unit cells. The objective is to color a consecutive (or block) cells in the grid with black color in each row and column that corresponds to the clues. Answer Set Programming (ASP) has been used to solve NP-complete problems. Answer set programming (ASP) is a new programming language paradigm combining the declarative aspect with non-monotonic reasoning. It provides a powerful language for a logical formulation of NP complete problems. Its nondeterministic computation liberates the user from the tree-search programming. For pixel puzzle it is possible to solve normal and big sized puzzles very fast. It turns out that, although answer set greatly simplifies the problem statement, it is comparable in efficiency to specialized programs.

Keywords: *Knowledge Representation, Answer Set Programming, Pixel Puzzle, Nonogram.*

I. INTRODUCTION

Nowadays, the logic type puzzles (such as Sudoku, Pixel puzzle, Kakuro, Divide-by-squares, etc.) are quickly spreading over the world, and there are many companies involved in the process of commercializing these puzzles [1, 2, 3]. In addition, puzzles and logic games in general are also sold to be played as games for mobile telephones or as web pages on the Internet, in a market with a turnover of millions of dollars per year.

In this work, we present one of the logic type puzzles, specifically the one known as Pixel Puzzle. Pixel puzzle is also called Nonograms or picture-logic or paint by numbers or Japanese puzzle. Pixel puzzles are logical puzzle games popular in Japan and Netherland. It was developed by Nikoli, the same company that created Sudoku. It has been studied mathematically, and it is known to be NP-complete [4], making search-based solution techniques practical only for small problems. Pixel puzzle consists of rectangular shape of any size, which is subdivided into unit cells. For each row and column, row clues and column clues are given. The row clues are on the left of each row and the column clues are on the top of each column. Each clue associated with a row or column indicates the length of the consecutive segments (or block) of black cells. The objective of the Pixel puzzle is to fill consecutive cells (or block cells), with black color, in each row and column; so that the filled consecutive cells length and sequence corresponds to the clues. In addition, there must be at least one empty square between adjacent blocks. Usually, the result of filled cells forms an image. Fig. 1(a) shows a simple Pixel puzzle example. The positive integers in the top of a column or left of a row stand for the lengths of block (consecutive) black cells in the column or row respectively. The puzzle in Fig. 1(a) has a unique solution shown in Fig. 1(b). However, some Pixel puzzles may have no solution, exactly one solution or many solutions. A clue of (e_1, …, e_k) means we shade k blocks of e_1, …, e_k cells, with at least one empty space between two consecutive blocks (e_i and e_{i+1}, $0 < i < k$). There may or may not be empty cells before the block of e_1 and after that of e_k.

	2	2 1 2	7	1 3	2 1
2					
2,1					
1,1					
3					
1,1					
1,1					
2					
1,1					
1,2					

(a) Pixel Puzzle Example

	2	2 1 2	7	1 3	2 1
2		■	■		
2,1		■	■		■
1,1			■		■
3		■	■	■	
1,1	■		■		
1,1	■		■		
2			■		
1,1		■		■	
1,2		■		■	■

(b) Solution of (a)

Figure 1, Pixel Puzzle Example and its Solution

In general, a pixel puzzle has a size of m-by-n, with m rows and n columns. For example, figure 1(a) has a size of m=9 and n=5.

Pixel puzzle started to appear weekly in Japanese newspaper *The Sunday Telegraph* in 1990. In the United Kingdom the name given to this kind of puzzle was Nonogram. In this paper, we are going to use pixel puzzle and Nonogram interchangeably. In the last few years the popularity of these puzzles has increased a lot. There are several companies which publish magazine and web pages exclusively devoted to Pixel puzzle in countries such as Spain, Germany, Italy or Finland.

There can be considerable differences in the difficulty level of Pixel puzzle. On the one hand, the Pixel puzzle that appear in newspapers can typically be solved with elementary reasoning and without complex deductions

which is by applying a series of simple logical rules, each of which considers only a single horizontal or vertical line. Later on we will refer to them as being simple. These puzzles will always have a unique solution. On the other hand, large random puzzles can be very difficult to solve, even using a computer, and may have many different solutions. Clearly, the fact that solving Pixel puzzle is NP-hard indicates that not all puzzles can be solved using simple logic reasoning.

Pixel puzzle application is not only confined to education/recreational use [5, 6]. They can be found in fields such as pattern recognition, where they are used to recognize printed letters [7] and sentences [8].

Logic Programming under answer set semantics provides a powerful language for a logical formulation of NP complete problems. Its nondeterministic computation liberates the user from the tree-search programming. Answer set programming (ASP) is a form of declarative programming that is emerged from logic programming with negation and reasoning formalism that is based on the answer set semantics [9, 10]. ASP is considered in the late 1990s as a new programming paradigm [11]. Answer set programming languages has been used to solve many real life application problems, among them, production configuration [12], decision support for NASA shuttle controllers [13], synthesis of multiprocessor systems [14], reasoning tools in biology [15], team building [16], composition of Renaissance music [17], and many more. A number of solvers have been proposed, such as: smodels [18], dlv [19], cmodels [20], assat [21], and clasp [22].

In this paper we present an automatic solving of pixel puzzles using answer set programming. First a brief overview of Answer Set Programming and its semantics is given. Then, a formal definition of pixel puzzle is presented. After that, we present the solution to the pixel puzzle under Answer Set Programming. Finally, we present experimental results and conclusion.

Related work:

There are several papers which discuss how to solve the Pixel puzzle. Some related papers [23, 24] solved this problem using an evolutionary algorithm for discrete tomography. Although this algorithm is quite effective at solving Nonograms, it cannot be used to find all solutions, if more than one solution exists.

In [25], ad-hoc heuristics is implemented. In [26, 27], logic rules and depth first search algorithm are implemented and later presented a comparison between the two approaches [28], the result was for smaller size Nonogram puzzles, DFS was faster, but for larger Nongram puzzles, GA was faster. However, GA may stuck in local optima. Other solvers convert the problem to an instance of a constraint satisfaction problem (CSP) such as integer programming [29, 30], or SAT [31, 32]. In [35] neural network were used to solve the pixel puzzle. Additional solvers can also be found in the internet [33, 34]. The related problem of generating Nonograms that are uniquely solvable is discussed in [35].

II. BRIEF OVERVIEW OF ANSWER SET PROGRAMMNG

We briefly recall the basics about ASP. An ASP-program is a collection of rules of the form

$$a_0 \leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, a_n \qquad (1)$$

Where, each a_i is an atom. The head of the rule is a positive atom which is the left hand side of the clause in (1). The body of the rule is composed of literals (a literal is an atom or its negation, denoted by *not a*) which is on the right side of the clause in (1). A rule without body is a fact. A rule without head is a constraint. Also, the rules can be positive (m>0); negative (n>0) or both (m>0 and n>0). The symbol *not* stands for default negation, also known as negation as failure.

If P is a ground, positive program (no negation as failure used), a unique answer set is defined as the smallest set of literals constructed from the atoms occurring in program P (minimal model). The last definition can be extended to any ground program P containing negation by considering the reduct of P with respect to a set of atoms X obtained by the Gelfond-Lifshitz's operator [9]. The reduct, P^X, of P relative to X is the set of rules:

$$a_0 \leftarrow a_1, \ldots, a_m$$

for all rules (1) in P such that $a_{m+1}, \ldots, a_n \notin X$. Then P^X is a program without the negation *not*. Then X is an answer set for P if X is an answer set for P^X.

Once a program is described as an ASP-program P, its solutions, if any, are represented by the answer set of P. One important difference between ASP semantics and other semantics is that a logic program may have several answer sets or may have no answer set at all.

Answer Set Programming is a totally declarative language. ASP programs are not algorithms describing how to solve the problem; the program is just a formal description of the problem. The solution is completely found by the solver. An ASP solver requires grounded programs as input, and that is why before searching the answer set or solutions, the program is grounded by a preprocessor. The computation of answer sets is done in two phases: (i) grounding of the logic program (P): that is eliminating variables to obtain a propositional program ground(P). (ii) Computation of answer sets on the propositional program ground(P).

III. PIXEL PUZZLE FORMAL DEFINITION

We begin by defining a single line (row or column) of a pixel puzzle. Let $\Gamma = \{0, 1, x\}$. The symbol "0" represents white pixel in the puzzle, the symbol "1" represents black pixel in the puzzle and the symbol "x" indicates an unknown color pixel (undecided) in the puzzle. Let $\Sigma = \{0, 1\} \subseteq \Gamma$. For w>0, define Γ^w (resp. Σ^w) to denote the set of all strings of length w over Γ (resp. Σ). Therefore, we have 2^w (resp. 3^w) strings over Γ (resp Σ).

We denote by S_w the set of all strings of length w over Γ. If $s \in S_w$ then we write $s = [s_i]$ for i=0, 1, 2, …, w-1 (i.e. zero index string) and $s_i \in \Gamma$. We let s_i=1 to mean the cell is shaded with black color, s_i=0 to mean the cell is shaded with white color and s_i=x if the cell is undecided color. A single line of pixel puzzle (row or column) is represented by a string $s \in S_w$.

A *clue* e^k of length k>=0 is an ordered series $(e_1, e_2, \ldots, e_k)$ where $e_j \in \{0, 1, 2, \ldots\}$ for $j=\{1, 2, \ldots, k\}$. Without abuse of notation, we consider that $e_i \in e^k$ to denote that e_i is among the ordered series of e^k. Each $e_i \in e^k$ (i=1, …, k) is called a clue block. If k=0, then we have the empty clue ε. For example, the first row of the pixel puzzle in figure 1(a) has a clue of (2) and the second row has a clue of (2, 1). Let E_k denote all the (infinite) set of all clues of length k, and put $E=\bigcup_{k=0}^{\infty} E_k$, where E_0 denotes the empty clue ε. We denote the set of all clues by $E_{nonogram} \subseteq E$.

We will use the shortcut a^* to denote zero or more of $a \in \Gamma$ and a^+ to denote one or more of $a \in \Gamma$ (similar to the convention of regular expressions).

If we have a string s over Γ, if a string s does not contain any "x" symbol then string s is called a specification. A finite specification s of length w over Σ satisfies a clue $e^k = (e_1, e_2, \ldots, e_k)$ if it satisfies the regular expression $(0^*1^{e1}0^+1^{e2}0^+\ldots1^{ek}0^*)$. For example the specification s of length five satisfies the clue (2) if $s \in \{11000, 01100, 00110, 00011\}$ which can be represented by the regular expression $(0^*1^20^*)$. For a specification s that satisfies a clue $(q_1, \ldots, q_r)$ of length r, then the specification s has a length w that must be at least $\sum_{i=1}^{r} q_i$ + r-1. For example, for a clue c^1 = (2) (of length one). The specifications that satisfies this clue must have a length at least two defined as the regular expression $(0^*1^20^*)$, whereas a clue e^2 = (2, 1) of length 2 is satisfied by a specification of length at least 4 defined as $(0^*1^20^+1^10^*)$.

For a string s of length w, and a clue $e^k = (e_1, e_2, \ldots, e_k)$ of length k. The clue sum is defined as: $S^{ek} = \sum_{i=1}^{k} e_i$. Each clue block e_i has an earliest start index and latest end index in the string s of length w. We can define the earliest start index P^{ek} in string s as follows:

$$P^{ek}(e_i) = \begin{cases} 0 & i=1 \\ P^{ek}(e_{i-1}) + e_{i-1} + 1 & i>1. \end{cases} \quad \ldots (2)$$

Proof: Given a clue e^k, then the first clue block e_1 starts at index 0. The rest clue blocks e_i (i>1), the earliest start index is e_{i-1} earliest start index plus the length e_{i-1} plus one for the space between block cells. □

The latest end index Q^{ek} in the string s of length w may be determined by:

$$Q^{ek}(e_i) = \begin{cases} w - S^{ek} + e_1 - k & i=1 \\ Q^{ek}(e_{i-1}) + e_i + 1 & i>1 \end{cases} \quad \ldots (3)$$

Proof: Given a clue e^k, then we need at least S^{ek} for the k clue blocks and k-1 for the spaces between every two consecutive clue blocks. Therefore, we need S^{ek}+k-1 cells. Since the string is of length w, then the latest end index of e_1 is w-S^{ek}-k+e_1.

The other clue blocks e_i (i>1), the latest end index is the e_{i-1} latest end index plus the length e_i plus one for the space between block cells. □

For example, given the second row clue e^2=(2, 1) and a string length of 5 as in Fig. 1(a), then S^{e2} = 3, $P^{e2}(2)$ = 0, $P^{e2}(1)$ = 0+2+1=3, $Q^{e2}(2)$ = 5-3+2-2=2, $Q^{e2}(1)$ = 2+1+1=4; see Fig. 2.

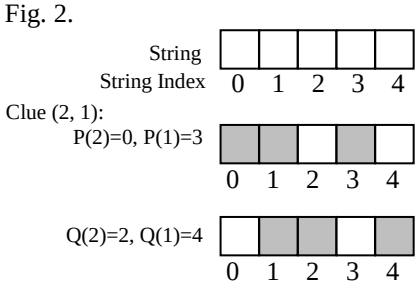

Figure 2, example of earliest start and latest end of a clue

A pixel puzzle is a two dimensional matrix N (mxn) of m-row strings (m>0) where each row string $r_i \in \Gamma^n$ (i=0, 1, …, n-1) and n-columns strings (n>0) $c_j \in \Gamma^m$ (j=0, 1, …, m-1), such that each row string r_i is associated with a clue $re_i \in E$ (i=0, 1, …, m-1) positioned on the left of each row and each column string c_j is associated with a clue $ce_j \in E$ (j=0, 1, …, n-1) positioned on the top of each column. A pixel puzzle matrix N is called complete filling if each row string r_i is a satisfiable specification with respect to clue re_i (i=0, 1, …, m-1) and a column string c_j is a satisfiable specification with respect to clue ce_j (j=0, 1, …, n-1). A complete filling pixel puzzle matrix is called a solution if $r_i[j] \oplus c_j[i] = 0$ for i=0, 1, …, m-1 and j=0, 1, …, n-1, where ⊕ is the normal bit exclusive-or operation and each re_i and ce_j satisfies their corresponding associated clues.

IV. PROBLEM DESCRIPTION AND RESOLUTION IN ASP

In this section, we describe Pixel puzzle problem in the language of gringo which is the grounder for the answer set programming solver. Initially, we have a matrix of (mxn) and all rows strings satisfy the regular expression (x^n) and all column strings satisfy the regular expression (x^m). Assume we have a pixel puzzle of m-rows and n-columns matrix. To solve the pixel puzzle, the following conditions must be satisfied:

N1) For each row string or column string s of length w, s[i] is either 0 or 1 for all i=0, …, w-1.

N2) For each clue $e^k = (e_1, \ldots, e_k)$ associated with a row string or column string s of length w, if the block of black cells asscoiated with e_i (for i=1, …, k-1) starts at index s[p] (for $p=P^{ek}(e_i), \ldots, Q^{ek}(e_i)-e_i$) and end at $s[p+e_i-1]$ and the block of black cells associated with e_{i+1} starts at index s[q] (for $q=P^{ek}(e_{i+1}), \ldots, Q^{ek}(e_{i+1})-e_{i+1}$) and end at $s[q+e_{i+1}-1]$, then $q>p+e_i$. Note that the inequality includes that there is at least one space between the block of black cells associated with e_i and the block of black cells associated with e_{i+1}.

N3) for each row string s1 and column string s2; $s1[i] \oplus s2[i] = 0$, where ⊕ is the normal exclusive-or.

A. Constructing the Data Module of D_1 of ASP

This module defines an instance of the Nonogram puzzle. This module consists of defining the Nonogram matrix (mxn) and a list of row and column clues associated with each row string and column string. The size of the Nonogram matrix is defined as:

```
rows(0..ROWMAX-1).
cols(0..COLMAX-1).
```

The constants ROWMAX and COLMAX are defined by the user.

The list of row clues are defined as set of facts of the form:

```
rowClue(row_number, block_length, index)
```

This fact defines the row clue. For each row number "row_number" in the pixel puzzle, the clue length is specified by the "block_length" and its order index is specified by "index". For example, the second row clues in figure (1) can be defined as:

```
rowClue(2, 2, 1).
rowClue(2, 1, 2).
```

Similarly, the column clue is defined as:

```
colClue(col_number, block_length, index).
```

Note that a row or a column that do not have a clue, then either we define the fact with block_length is set to zero or we do not define the fact at all.

B. The Pixel Puzzle Preparation Module D_2

This module defines new predicates that will simplify and speeding up finding the answer set models of the pixel puzzle logic. This module consists of the following rules:

- The first group consists of finding the sum of the clues for each row clue and column clue. It is suffice to write the following rules:

```
rowClueSum(R, RS) :- RS = #sum { B,I :
      rowClue(R, B, I) }, rows(R).
colClueSum(C, CS) :- CS = #sum { B,I :
      colClue(C, B, I) }, col(C).
```

The first rule is the sum of the row clue blocks, and the second rule represents the sum of the column clue blocks.

- The second group consists of finding the number of clue blocks in each row and column. This is done in ASP as:

```
rowClueCount(R, RC) :-
  RC = #count {I : rowClue(R, B, I) }, rows(R).
colClueCount(C, CC) :-
  CC = #count {I : colClue(C, B, I) }, cols(C).
```

The first rule finds the number of row clues and the second rule finds the number of column clues.

- The third group is to compute the earliest start index and the latest end index of each clue block using the formulas (2) and (3). It is suffice to write the following rules for the row clue:

```
rowClueRange(R, B, 1, 0, CE) :-
      rowClue(R, B, 1),
      rowClueSum(R, RS),
      rowClueCount(R, RC), B>0,
      CE = ROWMAX  – (RS + RC) + B
rowClueRange(R, B, I, CS, CE) :-
      rowClue(R, B, I), I>1, B>0,
      rowClueRange(R, B2, I-1, CS2, CE2),
      CS = CS2 + B2 + 1, CE = CE2 + B + 1.
```

Similarly for the column clue range are defined as:

```
colClueRange(C, B, 1, 0, RE) :-
      colClue(C, B, 1),
      colClueSum(C, CS), colClueCount(C, CC),
      RE = COLMAX  – (CS+CC) + B.

colClueRange(C, B, I, RS, RE):-
      colClue(C, N, I), I>1,
      ccolClueRange(C, B2, I-1, RS2, RE2),
      RS = RS2 + B2 + 1, RE = RE2 + B + 1.
```

Note, the rules defined in this module are all facts and it accelerates the search for a solution significantly.

C. The Pixel Puzzle Solver Module D_3

This module describes solving the pixel puzzle. The transition diagram of D_3 will be described by group of axioms:

- The first group defines the executability conditions for actions. We have two actions:
  1. "rowClueSelect(R, C, B, I)": which means that the clue block at row R with clue block length of B and order index I starts at index C.
  2. "colClueSelect(R, C, B, I)" :- which means the clue block at column C with clue block length of B and order index I starts at index R.

The rules are as follows:

```
1 {rowClueSelect(R, C, B, I):cols(C),
      C>=C1, C<=C2 } 1 :-
   rowClueRange(R, B, I, C1, C2),
   B>0.
1 { colClueRange(R, C, B, I): row(R),
      R>=R1, R<=R2 } 1 :-
   colClueRange(C, B, I, R1, R2),
   B>0.
```

The actions "rowClueSelect" and "colClueSelect" selects one starting index available for each clue block. The rule is a choice rule that is bounded by 1 and 1. This means it will select only one start index for a clue block. This rule is the generate rule that will generates all possible selection of clue indices. The rules will execute when clue length is greater than zero.

- The second group contains the causal laws describing direct effect of actions. When an action "rowClueSelect(R, C1, B, I)" selects a clue block with length B and order I at a specified index C1 then the following will happen:
  1. The cell indices from (R, C1) to (R, C1+B-1) are set to one (i.e. color as black).
  2. The cell indices at (R, C1-1) must be zero (i.e. colored as white).
  3. The cell indices at (R, C1+B) must be zero (i.e. colored as white).
  4. If I=1, then all cells less that C1 must be zero (i.e. colored as white).
  5. If I is equal to the count of the clues associated with row R (this is already calculated in module D_2), then all cells greater that C1+B must be zero (i.e. colored as white).

  For example, it is suffice to have the rules to implement the above effects for a row clue selection action:

```
rowOnes (R, C) :-
    rowClueSelect(R, C1, B, I),
    rowClue(R, B, I),
    cols(C1), cols(C),
    C>=C1, C<C1+B
zero(R,C+B) :-
    rowClueSelect(R, C, B, I),
    rowClue(R, B, I),
    col(C), col(C+B).
zero(R,C-1) :-
    rowClueSelect(R, C, B, I),
    rowClue(R, B, I),
    col(C), C>0.
zero(R, C) :-
    rowClueSelect(R, C1, B, 1),
    rowClue(R, B, I),
    col(C), col(C1), C<C1.
zero(R, C) :-
    rowClueSelect(R, C1, B, Cr),
    rowClue(R, B, I),
    col(C), col(C1),
    rCountClues(R, Cr), C>Cr.
```

  The first to the fifth rules defined implements the steps 1-5 above respectively. Similar set of rules are written for the column clue selection action.

- The third group of rules are the constraints that eliminate unwanted answer set models, which are defined as follows:

```
:- rowOnes(R, C), not colOnes(R, C).
:- colOnes(R, C), not rowOnes(R, C).
```

  The first constraint rejects all answer sets with ones set by row select action and not selected by column select action. The second constraint rejects answer sets with ones set by column select action and not selected by row select action.

  These two constraints is an implementation of the exclusive-or of rows and columns (condition N3).

  Other constraints are needed that will eliminate all answer sets that set a cell with one (black color) and zero (white color). These rules are as follows:

```
:- rowOnes(R,C), zeros(R, C).
:- colOnes(R, C), zeros(R, C).
```

  These rules are implementation of condition N1.

  The last constraints rejects all answer set that violate condition N2. The constraint is defined as:

```
:- rowClueSelect(R, C, B, I),
    rowClueSelect(R, C1, B1, I+1), C+B>=C1,
    rowClue(R, B, I), rowClue(R, B1, I+1),
    cols(C1), cols(C).
:- colClueSelect(R, C, B, I),
    colClueSelect(R1, C, B1, I+1), R+B>=R1,
    colClue(C, B, I), colClue(C, B1, I+1),
    rows(R1), rows(R).
```

Before the selection process, there are logic rules that may be applied to reduce the search space of selection process. These logic rules will determine the zeros (white color) and ones (black color) in the row and column strings. The rules are defined as:

1) Given a clue, if the difference between the earliest start index and the latest end index are equal to the clue length, then this clue must start at the earliest start index in the row string or column string. This is defined as:

```
rowClueSelect(R, C) :-
    rowClueRange(R, B, I, C, C+B-1), B>0.
colClueSelect(R, C ) :-
    colClueRange(C, B, I, R, R+B-1), B>0.
```

2) If a clue length is 0, then the string associated with that clue has all "0" symbol" (white color). This is defined as:

```
zero(R, C) :- rowClue(R, 0, 1), cols(C).
zero(R, C) :- colClue(C, 0, 1), rows(R).
```

3) If a clue is not defined, then the string associated with that clue has all "0" symbol (i.e. white color). This is defined as:

```
rClueExist(R) :- rowClue(R, B, I), B>0.
cClueExist(C) :- colClue(C, B, I), B>0.
zero(R, C) :- not rClueExist(R), cols(C),
                rows(R).
zero(R, C) :- not cClueExist(C), cols(C),
                rows(R).
```

   The first two rules check if a clue is defined for a row or column. The second two groups set the symbol "0" (i.e. white color) if a clue if not defined for a row or a column.

4) Given a clue $e^k = (e_1, \ldots, e_k)$, a common squares in the range of earliest start index and latest end index should be set to symbol "1". This is defined as:

```
rowOnes(R, C) :-
        rowClueRange(R, B, I, CS, CE),
        C<CS+B, C>CE-B.
colOnes(R, C) :-
        colClueRange(C, B, I, RS, RE),
        R<RS+B, R>RE-B.
```

V. EXPERIMENTAL RESULTS

Our experiments were designed to assess the performance of each of the ASP on pixel puzzle problems. We used the webpbn bench mark. These benchmarks have different pixel puzzle sizes ranges from 10x5 to 95x80, to 59x99.

Table I shows running the answer set program on several instances of the pixel puzzle problem. All solvers in Table I were run on An Intel core 2 due laptop with 1.2 GHz processor and 4GB RAM is used.

The benchmarks problems of varying sizes are used to compare some of the well known pixel puzzle solvers with our solver. The results are presented in Table I, where the '+' sign means computation stayed for 8-hours and did not complete. The size of each problem is written under the name of the problem. The table compares the computation time in seconds of each problem using different solvers. The shape column is the solution of the problem where the hidden image is revealed. A detail of each of the pixel puzzle solvers presented in Table I is summarized in [34].

TABLE I. Benchmark Results

Problem	Copris	Wu	Wolter	Olsak	Simpson	Our code	Shape
#1 Dancer 5x10	0.61	0.00	0.00	0.00	0.00	0.031	
#6: Cat 20x20	2.33	0.00	0.00	0.00	0.00	0.062	
#21 Skid 14x25	2.24	0.00	0.00	0.00	0.00	0.047	
#27 Buck 27x23	3.87	0.00	0.00	0.00	0.00	0.140	
#23 Edge 10x11	0.60	0.00	0.00	0.00	0.00	0.031	
#2413 Smoke 20x20	3.17	0.00	0.00	0.00	0.00	0.078	
#16 Knot 34x34	5.43	0.00	0.00	0.00	0.00	0.874	
#529 Swing 45x45	5.74	0.00	0.00	0.00	0.00	0.764	
#65 Mum 34x40	5.43	0.00	0.01	0.00	0.01	0.530	
#7604 DiCap 52x63	10.37	0.01	0.02	0.00	0.00	3.869	
#1694 Tragic 45x50	6.11	0.02	0.04	0.03	0.54	1.435	
#1611 Merka 55x60	8.42	0.02	0.01	0.01	0.00	2.293	
#436 Petro 40x35	5.05	0.01	0.06	15.20	0.10	0.546	
#4645 M&M 50x70	6.90	-	0.07	0.10	0.02	2.231	
#3541 Signed 60x50	7.37	0.07	0.04	1.10	324.00	2.278	
#803 Light 50x45	4.71	0.20	0.38	+	0.02	0.827	
#6574 Forever 25x25	3.70	0.34	3.70	2.00	18.90	0.156	
#10810 Center 60x60	12.94	5.40	8.60	0.00	0.01	4.898	
#2040 Hot 55x60	8.90	0.11	0.83	+	72.00	3.916	
#6739 Karate 40x40	6.44	0.06	0.80	17.30	1098.00	0.952	
#8098 9-Dom 19x19	3.01	20.30	11.00	240.00	+	0.156	
#2556 Flag 65x45	39.07	-	0.55	1.50	0.00	0.874	
#2712 Lion 47x47	11.29	53.40	6.30	+	+	12.730	
#10088 Marley 52x63	15.49	54.00	+	+	+	7.114	
#18297 Thing 36x42	26.03	504.00	402.00	+	+	12.402	
#9892 Nature 50x40	38.58	108.00	+	1116.00	+	43.696	
#12548 Sierp 47x40	49.19	+	+	+	+	283.329	

We see from Table I, that our code solved all the available benchmarks. However, many solvers available, except for Copris, could not solve all the benchmarks. The ASP code is comparable in efficiency to the specialized software available. For some of the problems our code is faster than other solvers such as: #18297 Thing. In comparison with the Copris solver, some problems were solved faster in our code than the Copris solver such as: #6 Cat, #21 Skid, #27 Buck, …. However, ASP code could not solve the problem fasse, meow, and knotty.

VI. CONCLUSION AND FUTURE WORK

In this paper, we present an approach that uses ASP to represent the pixel puzzle problem to reveal the hidden picture. Pixel puzzle is known to be a NP-Complete problem. We have proposed to investigate and evaluate the capabilities of ASP to solve the pixel puzzle logic problem. The program presented in this paper was able to solve some problems efficiently than many other solvers, such as Forever, Flag, and Marlay. ASP is expressive enough to represents the constraint of the pixel puzzle problem. The paper also shows the expressive use of the aggregates and optimization sentences defined in the 'clingo' solver. In the future, we can try to experiment with

more logical reasons to increase the time efficiency of the ASP solver.

REFERENCES

[1] Conceptis Puzzles Inc., http://www.conceptispuzzles.com

[2] Catchy Software Inc., http://www.catchysoft.com/jcwd.html

[3] Nikoli Software Inc., http://www.nikoli.co.jp/en/

[4] N. Ueda, and T. Nagao, "NP-completeness results for Nonogram via parsimonious reductions," Technical Report TR96–008, Department of Computer Science, Tokyo Institute of Technology, 1996.

[5] S. Salcedo-sanz, J.A. Portilla-Figueras, E.G. Ortiz-Garcia, A. M. Perez-Bellido, and X. Yao, "Teaching advanced features of evolutionay algorithms using Japanese puzzles", IEEE Trans., Educ. 50(2), 2007, pp. 151-156.

[6] J.T. Tsai, P.Y. Chou, and J.C. Fang, "Learning intelligent genetic algorithms using Japanese nonograms", IEEE Trans. Educ. 99, 2011, pp. 164–168.

[7] Y.S. Sohn, K.S. Oh, and B.S. Kim, "A recognition method of the printed alphabet by using nonogram puzzle," in: Proceedings of the 8th International Symposium on Advanced Intelligent Systems, 2007, pp. 232–236.

[8] Y.S. Sohn, "Recognition of the printed English sentence by using Japanese puzzle", Int. J. Fuzzy Logic Intell. Syst. 8, 2008, pp. 225–230.

[9] M. Gelfond and V. Lifschitz, The Stable Model Semantics for Logic Programming, ICLP/SLP, 1988, pp. 1070-1080.

[10] C. Baral. "Knowledge Representation, Reasoning and Declarative Problem Solving," Cambridge University Press, 2003.

[11] V. Marek and M. Truszczyński, "Stable models and an alternative logic programming paradigm," In Apt, Krzysztof R. The Logic programming paradigm: a 25-year perspective, Springer. pp. 169-181, 1991.

[12] T. Soininen and I. Niemela, "Developing a declarative rule language for applications in product configuration, " In Gupta, G., ed.: Proceedings of the First International Workshop on Practical Aspects of Declarative Languages (PADL'99), pp. 305–319, Springer 1999.

[13] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry, "An A-prolog decision support system for the space shuttle," *Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pp 169-183, Springer-Verlag, 2001.

[14] H. Shebabi, P. Mahr, C. Bobda, M. Gebser, and T. Schaub, "Answer set vs integer linear programming for automatic synthesis of multiprocessor systems from real-time parallel programs," Journal of Reconfigurable Computing, 2009.

[15] M. Gebser, T. Schaub, S. Thiele, and P. Veber, "Detecting inconsistencies in large biological networks with answer set programming," Theory and Practice of Logic Programming 11 (2), pp1–38, 2011.

[16] G. Grasso, S. Iiritano, N. Leone, V. Lio, F. Ricca, and F. Scalise, "An ASP-based system for team-building in the Gioia-Tauro seaport". In Proceedings of the Twelfth International Symposium on Practical Aspects of Declarative Languages (PADL'10), Volume 5937 of Lecture Notes in Computer Science., Springer-Verlag, pp. 40–42, 2010.

[17] G. oenn, M. Brain, M. de Vos, and J. Fitch, "Automatic composition of melodic and harmonic music by answer set programming". Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08). Volume 5366 of Lecture Notes in Computer Science., pp. 160–174, Springer-Verlag 2008.

[18] P. Simons, I. Niemels, and T. Soininen, "Extending and implementing the stable model semantics". Artificial Intelligence 138 (1-2), pp. 181–234, 2002.

[19] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. "The DLV system for knowledge representation and reasoning," ACM Transactions on Computational Logic, 7(3):499–562, July 2006.

[20] Yu. Lierler and M. Maratea, "Cmodels-2: SAT-based answer set solver enhanced to non-tight programs," In Proc. of LPNMR-7, 2004.

[21] F. Lin and Yu. Zhao, ASSAT: "Computing answer sets of a logic program by SAT solvers," Artificial Intelligemce 157(1-2), pp. 115-137, 2014.

[22] M. Gebser, B. Kaufmann, A. Neumann and T. Schaub, "clasp: A Conflict-Driven Answer Set Solver," LPNMR'07, 2007.

[23] KJ. Batenburg, "An evolutionary algorithm for discrete tomography," Master thesis in computer science, University of Leiden, The Netherlands, 2003.

[24] K. Batenburg, and W. Kosters, "A discrete tomography approach to Japanese puzzles," *Proceedings of the 16th Belgium-Netherlands Conference on Artificial Intelligence (BNAIC*, 2004, pp. 243–250.

[25] S. Salcedo-Sanz, E. Ortiz-García, A. Pérez.Bellido,J. Portilla-Figueras, and X. Yao, "Solving Japanese puzzles with heuristics," In: IEEE symposium on computational intelligence and games, Honolulu, USA, April 2007.

[26] M. Q. Jing, C. H. Yu, H. L. Lee and L. H. Chen , "Solving Japanese puzzles with logical rules and depth first search algorithm" , Proceedings of the 2009 International Conference on Machine Learning and Cybernetics , vol. 5 , pp.2962 -2967 , 2009

[27] C.H. Yu, L.H. Lee, and L.H. Chen, "An efficient algorithm for solving nonograms," *Applied Intelligence*, *35*(1), 2011, pp. 18–31.

[28] W. A. Wiggers, "A comparison of a genetic algorithm and a depth first search algorithm applied to Japanese nonograms," Twente student conference on IT, Jun 2004

[29] R.A. Bosch, "Painting by numbers", Optima 65 2001, pp. 16–17.

[30] L. Mingote, and F. Azevedo, "Colored nonograms: an integer linear programming approach," in: Proceedings of the Portuguese Conference on Artificial Intelligence, 2009, pp. 213–224.

[31] K.J. Batenburg and W.A. Kosters, "Solving nonograms by combining relaxations", Pattern Recognition 42(8), 2009, pp. 1672–1683.

[32] A. Metodi, M. Codish, V. Lagoon, and P.J. Stuckey, "Boolean equi-propagation for optimized SAT encoding", in: Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming, 2011, pp. 621–636.

[33] S. Simpson, Website nonogram solver, http://www.comp.lancs.ac.uk/~ss/nonogram/links.html.

[34] Wolter, Survey of paint-by-numbers puzzle solvers, http://webpbn.com/survey/index.html.

[35] J.K. Vis, W.A. Kosters, and K.J. Batenburg, "Discrete tomography: A neural network approach," In Proceedings of the 23rd Benelux Conference on Artificial Intelligence (BNAIC), pages 328–335, 2011.

[36] E. G. Ortiz-García, S. Salcedo-Sanz, J.M. Leiva-Murillo, Á.M. Pérez-Bellido, and J.A. Portilla-Figueras, "Automated generation and visualization of picture-logic puzzles", Comput. Graph., 31 2007, pp. 750–760.