An Improved Sub-optimal Algorithm for Solving $(N^2 - 1)$-Puzzle

Pavel Surynek[1,2] and **Petr Michalík**[1,3]

[1]*Charles University in Prague*
Faculty of Mathematics and Physics
Department of Theoretical Computer Science and Mathematical Logic
Malostranské náměstí 25, Praha, 118 00, Czech Republic

[2]*Kobe University*
Graduate School of Maritime Sciences
Division of Maritime Management Sciences
5-1-1 Fukae-minamimachi, Higashinada-ku, Kobe 658-0022, Japan

[3]*Accenture Central Europe B.V. Consulting*
Jiráskovo náměstí 1981/6, Praha, 120 00, Czech Republic

pavel.surynek@mff.cuni.cz, petr.michalik@accenture.com

Abstract. The problem of solving $(n^2 - 1)$-puzzle sub-optimally in the on-line mode is addressed in this manuscript. The task is to rearrange $n^2 - 1$ pebbles on the square grid of the size of $n \times n$ using one vacant position to a desired goal arrangement. A new polynomial-time algorithm is proposed and experimentally analyzed. The new algorithm is trying to move pebbles in more efficient way than existent algorithms by grouping them into so-called snakes and moving them jointly within the snake. An experimental evaluation showed that our new snake-based algorithm produces solutions that are 8% to 9% shorter than solutions generated by the existent state-of-the-art algorithm.

Keywords: $(n^2 - 1)$-puzzle, 15-puzzle, on-line algorithm, polynomial complexity, multi-robot path planning, multi-agent coordinated motion.

1. Introduction and Motivation

The $(n^2 - 1)$-puzzle [1, 4, 5, 6, 18] represents one of the best known examples of *relocation* problem. It is important both practically and theoretically. From the theoretical point of view it is interesting for the hardness of its optimization variant which is known to be NP-hard [5, 6]. Practically it is important since many real-life relocation problems can be solved by techniques developed for $(n^2 - 1)$-puzzle. Those include *multi-robot path planning* [8, 9, 10, 11], *rearranging* of shipping containers in warehouses, or *coordination* of vehicles in dense traffic. Moreover, the reasoning about relocation/coordination tasks should not be limited to physical entities only. Many tasks such as planning of *data transfer*, *commodity transportation*, and *motion planning* of units in *computer-generated imagery* can be tackled using techniques originally developed for $(n^2 - 1)$-puzzle.

In this manuscript, we concentrate ourselves on solving $(n^2 - 1)$-puzzle sub-optimally in the on-line mode, that is by fast polynomial-time algorithm. We are trying to improve the basic incremental placing of pebbles as it is done by the existent on-line solving algorithm of Parberry [4] by moving them in groups called *snakes*. Moving pebbles jointly in snakes is supposed to be more efficient in terms of the total number of

moves than moving them individually as it was originally proposed [4]. A new algorithm exploiting snake-line movements is presented. An extensive competitive experimental evaluation was done to evaluate qualities of the new algorithm.

The presented work originates in the master thesis of the second author [3]. The manuscript of organized as follows. The problem of (n^2-1)-puzzle is formally introduced in Section 2. An overview of existent solving algorithm and other related solving approaches is given in Section 3. The main part of the paper is constituted by Section 4 and Section 5 where the new algorithm is introduced and its experimental evaluation is given respectively.

2. Problem Statement

The (n^2-1)-puzzle consists of a set of pebbles that are moved over a square grid of size $n \times n$ [1, 4, 5, 6, 18]. There is exactly one position vacant on the grid and others are occupied by exactly one pebble. A pebble can be moved to the adjacent vacant position. The task is to rearrange pebbles on the grid into a desired goal state. The formal definition follows.

2.1. *Formal Definition*

Sets of pebbles, we will be working with, will be denoted as Ω_n for $n \in \mathbb{N}$. It holds that $|\Omega_n| = n^2 - 1$ for every $n \in \mathbb{N}$. It is supposed that pebbles from a set Ω_n are arranged on a square grid of the size $n \times n$ where each pebble is placed into one of the cells of the grid. There is at most one pebble in each cell of the grid; one cell on the grid remains always vacant. See Figure 1 for illustration.

Definition 1 ***(arrangement in a grid).*** An *arrangement* of a set of pebbles Ω_n in a square grid of the size $n \times n$ with $n \in \mathbb{N}$ is fully described using two functions $x_n: \Omega_n \longrightarrow \mathbb{N}$ and $y_n: \Omega_n \longrightarrow \mathbb{N}$ that satisfy the following conditions:

(i) $x_n(p) \in \{1,2,\dots,n\}$ and $y_n(p) \in \{1,2,\dots,n\}$ $\forall p \in \Omega_n$;

(ii) $|\{p \in \Omega_n | (x_n(p), y_n(p)) = (i,j)\}| \leq 1$ for $\forall i,j \in \{1,2,\dots,n\}$ (every cell of the grid is occupied by at most one pebble)

(iii) $\exists i,j \in \{1,2,\dots,n\}$ such that $\forall p \in \Omega_n$ $(x_n(p), y_n(p)) \neq (i,j)$ (there exists a cell in the grid that remains vacant).

For convenience, we will also use some kind of an inverse to x_n and y_n which will be called an *occupancy* function and denoted as $\sigma_n: \{1,2,\dots,n\} \times \{1,2,\dots,n\} \longrightarrow \Omega_n \cup \{\perp\}$. It holds that $\sigma_n(i,j) = p$ if and only if $\exists p \in \Omega_n$ such that $x_n(p) = i$ and $y_n(p) = p$ or $\sigma_n(i,j) = \perp$ if no such pebble p exists (that is, if the cell (i,j) is vacant). □

The arrangement of pebbles in the grid can be changed through moves. An allowed *move* is to shift a pebble horizontally or vertically from its original cell to the adjacent vacant cell. Formally, the notion of move is described in the following definition. Four

types of moves are distinguished here: *left*, *right*, *up*, and *down* – only left move is defined formally; *right*, *up*, and *down* moves are analogous.

Definition 2 ***(left move).*** A *left* move with pebble $p \in \Omega_n$ can be done if $x_n(p) > 1$ and $\sigma_n(x_n(p)-1, y_n(p)) = \perp$; it holds for the resulting arrangement after the move described by x'_n and y'_n that $x'_n(q) = x_n(q)$ and $y'_n(q) = y_n(q)$ $\forall q \in \Omega_n$ such that $q \neq p$ and $x'_n(p) = x_n(p) - 1$ and $y'_n(p) = y_n(p)$. □

We are now able to define the (n^2-1)-puzzle using the formal constructs we have just introduced. The task is to transform a given initial arrangement of pebbles in the grid to a given goal one using a sequence of allowed moves.

Definition 3 ***((n^2-1)-puzzle).*** An *instance of* (n^2-1)*-puzzle* is a tuple $(n, \Omega_n, x_n^0, y_n^0, x_n^+, y_n^+)$ where $n \in \mathbb{N}$ is the size of the instance, Ω_n is a set of pebbles, x_n^0 and y_n^0 is a pair of functions that describes the *initial arrangement* of pebbles in the grid, and x_n^+ and y_n^+ is a pair of functions that describes the *goal arrangement* of pebbles. The task is to find a sequence of allowed moves that transforms the initial arrangement into the goal one. Such sequence of moves will be called a *solution* to the instance. □

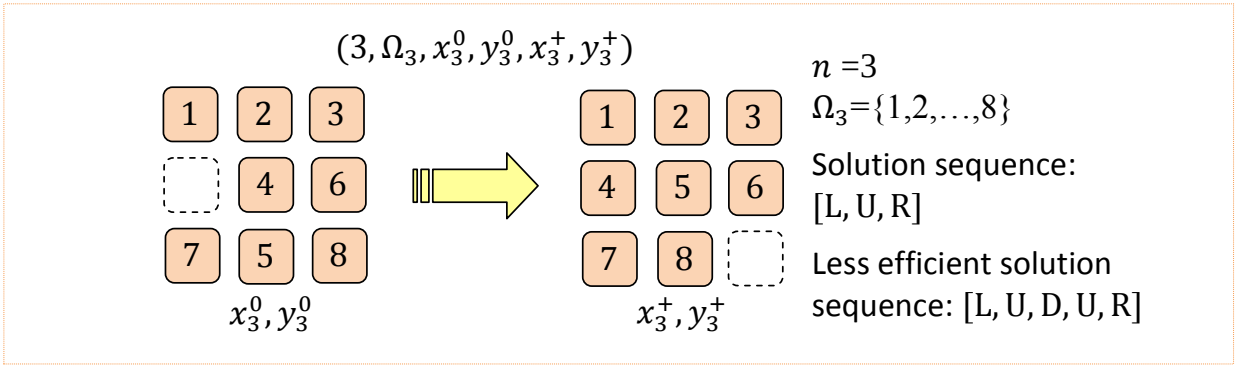


Figure 1. *An illustration of* (n^2-1)*-puzzle.* The initial and the goal arrangement of pebbles on the square grid of size 3 × 3 are shown. Two solutions of the instance are shown as well.

Again it is supposed that the occupancy function is available with respect to the initial arrangement x_n^0, y_n^0 and the goal arrangement x_n^+, y_n^+; that is, we are provided with occupancy functions σ_n^0 and σ_n^+. To avoid special cases it will be also supposed that $\sigma_n^+(n, n) = \perp$; that is, the vacant position is finally in the right bottom corner.

2.2. *Complexity and Variants of the Problem*

It is known that the decision variant of (n^2-1)-puzzle (that is, the yes/no question whether there exists a solution to the given instance) is in *P* [1, 4, 18]. It can be checked

by using simple parity criterion. Using techniques for rearranging pebbles over graphs [1] a solution of length $\mathcal{O}(n^6)$ can be constructed in the worst-case time of $\mathcal{O}(n^6)$ if there exists any. An approach dedicated exclusively to (n^2-1)-puzzle is [4] able to generate a solution of length $\mathcal{O}(n^3)$ in the worst-case time of (n^3) if there exists any.

If a requirement on the length of the solution is added, the problem becomes harder. It is known that the decision problem of whether there exists a solution to a given (n^2-1)-puzzle of at most the given length is NP-complete [6].

3. The Original Solving Algorithm and Related Works

A special sub-optimal solving algorithm dedicated for (n^2-1)-puzzle has been proposed by Parberry in [4]. As our new solving algorithm is based on the framework of the original one, we need to recall it at least briefly in this section.

3.1. *Algorithm of Parberry*

The algorithm of Parberry [4] sequentially places pebbles into rows and columns. More precisely, pebbles are placed sequentially into the first row and then into the first column, which reduces the instance to that of the same type but smaller – that is, we obtain a $((n-1)^2-1)$-puzzle.

Algorithm 1. *The original algorithm of Parberry for solving* (n^2-1)*-puzzle* [4]. The main loop of the algorithm is shown. Detailed description of placement of individual pebbles is not shown here – it will be discussed in the context of new approach for pebble placement.

---

procedure *Solve-N^2-1-Puzzle*$(n, \Omega_n, x_n^0, y_n^0, x_n^+, y_n^+)$

/* A procedure that produces a sequence of moves that solves the given (n^2-1)*-puzzle*.
Parameters: n, Ω_n - a size of the puzzle and a set of pebbles,
x_n^0, y_n^0 - an initial arrangement of pebbles in the grid,
x_n^+, y_n^+ - a goal arrangement of pebbles in the grid. */

1: $(x_n, y_n) \leftarrow (x_n^0, y_n^0)$
2: for $i = 1,2, \dots, n-3$ do
3: for $j = i, i+1, \dots, n$ do {current row is solved – from the left to the right}
4: $p \leftarrow \sigma_n^+(i,j)$
5: if $(i,j) \neq (x_n^+(p), y_n^+(p))$ then
6: $(x_n, y_n) \leftarrow$ *Place-Pebble*(x_n, y_n, i, j, p)
7: $\Omega_n \leftarrow \Omega_n \setminus \{p\}$
8: for $j = n, n-1, \dots, i+1$ do {current column is solved – from the bottom to the up}
9: $p \leftarrow \sigma_n^+(i,j)$
10: if $(i,j) \neq (x_n^+(p), y_n^+(p))$ then
11: $(x_n, y_n) \leftarrow$ *Place-Pebble*(x_n, y_n, i, j, p)
12: $\Omega_n \leftarrow \Omega_n \setminus \{p\}$
13: $\Omega_3 \leftarrow \Omega_n$; $x_3^0 \leftarrow x_n|_{\Omega_3}$; $y_3^0 \leftarrow y_n|_{\Omega_3}$; $x_3^+ \leftarrow x_n^+|_{\Omega_3}$; $y_3^+ \leftarrow y_n^+|_{\Omega_3}$ {restriction on Ω_n}
14: *Solve-8-Puzzle*$(\Omega_3, x_3^0, y_3^0, x_3^+, y_3^+)$ {the residual 8-puzzle is solved by A* algorithm}

This process of placement of pebbles is repeated until an 8-puzzle on the grid of size 3×3 is obtained. The final case of 8-puzzle is then solved optimally by the A* algorithm [7].

The main loop of the algorithm is shown in pseudo-code as Algorithm 1. The algorithm uses two high-level functions *Place-Pebble*, which conducts placement of a pebble to a given position, and *Solve-8-Puzzle*, which finalizes the solution by solving the residual 8-puzzle.

The placement of pebbles implemented within the function *Place-Pebble* will be discussed in more details later in the context of our improvement. Nevertheless, it is done quite naturally by moving a pebble first *diagonally* towards the goal position if necessary and then *horizontally* or *vertically*. To be able to conduct diagonal, horizontal and vertical movement a vacant position needs to be moved together with the pebble being placed. Actually, the vacant position is moving around the pebble always to the front in the direction of the intended move. After having vacant position in the front, the pebble is moved forward. It is necessary to avoid already placed pebbles when placing a new one.

3.2. ***Other Related Works***

The $(n^2 - 1)$-puzzle represents a special variant of a more general problem of *pebble motion problem on a graph* [1, 11, 18]. The generalization consists in the fact that there is an arbitrary undirected graph representing the environment instead of the regular grids as it is in the case of $(n^2 - 1)$-puzzle. There are also pebbles that are placed in vertices of the graph while at least one vertex remains vacant. The allowed state transition is a single move with a pebble to a vacant adjacent vertex. The task is expectably to rearrange pebbles from a given initial arrangement to a given goal one.

Although the problem has been studied long time ago [1, 18] recently there has been a considerable progress. The first new work showing solvability of every instance of pebble motion problem consisting of bi-connected graph [16, 17] containing at least two vacant positions is [10]. The related solving algorithm called *BIBOX* [10] can produce solution of length at most $\mathcal{O}(|V|^3)$ in the worst-case time of $\mathcal{O}(|V|^3)$ (V is the set of vertices of the input graph). The *BIBOX* algorithm also generates solutions that are significantly shorter than those generated by algorithms from previous works are [1, 18].

More results followed then. A generalization of *BIBOX* algorithm called *BIBOX-*θ is described in [11]. It does not need the second vacant position and again can solve instances on bi-connected graphs (notice that the grid of $(n^2 - 1)$-puzzle is a bi-connected graph; hence *BIBOX-*θ is applicable to it). Theoretically, it generates solutions of the worst-case length of $\mathcal{O}(|V|^4)$; however, practically solutions are much shorter.

Two years later an algorithm called *Push-and-Swap* has been published in [2] – it shows that for every solvable instance on arbitrary graph containing at least two vacant positions a solution of length $\mathcal{O}(|V|^3)$ can be generated.

In all the above results the solution length is sub-optimal and the worst-case time complexity is guaranteed (it is polynomial). A progress has been also made in optimal

solving of the pebble motion problem. A new technique that can optimally solve a special case consisting of a grid with obstacles and relatively small number of pebbles is described in [13]. It is based on an informed search, which however does not guarantee time necessary to produce a solution (the time may be exponential in the size of the instance).

Special cases of the problem with large graphs and relatively sparsely arranged pebbles are studied in [14, 15]. These new techniques are focused on applications in *computer games*. The complexity as well as the solution quality is guaranteed by these techniques. Another specialized technique for relatively large graphs and small number of pebbles has been developed within [8, 9]. The graph representing the environment is decomposed into subgraph patterns, which are subsequently used for more efficient solving by search.

Another closely related problem is known as *multi-robot path planning* [8, 9, 10, 11, 12] (or as *cooperative multi-agent path planning* or *path-finding* [14, 15]). It is a **relaxation** of pebble motion on a graph where simultaneous moves that do not conflict with each other are allowed in a single time step. Observe, that all the algorithms developed for multi-robot path planning applies to pebble motion problems and vice versa. Regarding the optimality of solutions, which in case of multi-robot path planning should be defined with respect to its makespan, the situation is again not very optimistic as it is shown in [12]. The decision version of the optimization variant of multi-robot path planning is NP-complete.

4. A New Solving Approach Based on 'Snakes'

In this section, we are about to define a new concept of a so-called *snake*. Informally, a snake is a sequence of pebbles that consecutively neighbors with a pebble that proceeds. As we will show, moving and placing a snake as a whole is much more efficient than moving and placing individual pebbles it consists of.

Recall that original algorithm for solving the puzzle [4] places pebbles individually into currently solved row or column. This may be inefficient if two or more pebbles that need to be placed are grouped together in some location distant from their goal location. In such a case, it is necessary that the vacant position is moved together with the pebble being placed and then it is moved back to the distant location to allow movement of the next pebble. If we manage all the pebbles forming the group to move from their distant location to their goal positions jointly, multiple movements of the vacant position between the distant location and goal positions may be eliminated.

4.1. *Formal Definition of a 'Snake'*

Consider a situation shown in Figure 2 where pebbles 1 and 2 are grouped together in a location distant from their goal positions. The original algorithm consumes $16n - 20$ moves to place both pebbles successfully to their goal positions. If pebbles are moved not

one by one but jointly as it is shown in Figure 3, much less movements are necessary. Grouping pebbles can save up to $4n$ moves.

This is the basic idea behind the concept of snake. Let us start with definition of a *metric* on the grid of the puzzle. Then the definition of the snake will follow.

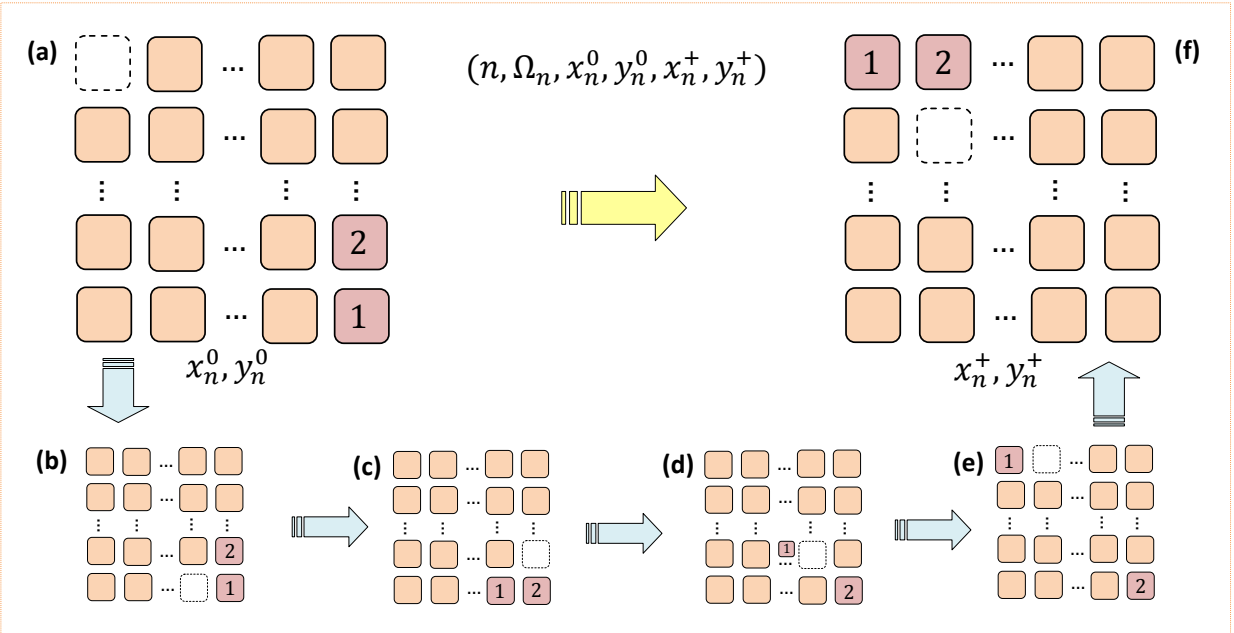


Figure 2. *A setup of* (n^2-1)*-puzzle where the original algorithm* [4] *is inefficient.* Pebbles 1 and 2 need to be moved from the bottom right corner (a) to the upper left corner (f). First, pebble 1 is moved diagonally to its goal position (b, c, d, and e). After pebble 1 is successfully placed, vacant position is moved towards pebble 2 and it starts to move in the same way as pebble 1 to its goal position. The whole process of rearranging consumes $16n-20$ moves.

Definition 4 ***(Manhattan distance).*** A *Manhattan distance* for the (n^2-1)-puzzle $\mu_n: \Omega_n \times \Omega_n \longrightarrow \mathbb{N}_0$ is a metric on the set of pebbles with respect to the arrangement of pebbles on the grid such that $\mu_n(p,q) = |x_n(p) - x_n(q)| + |y_n(p) - y_n(q)|$ for any two pebbles $p, q \in \Omega_n$. □

Having a metric on the grid of the puzzle, we are able to define neighborhood of a pebble (which will be a ring around a pebble in our case).

Definition 5 ***(Manhattan neighborhood).*** A *Manhattan neighborhood* of a pebble p denoted as $\nu(p)$ is a set of those pebbles that are located directly left, right, above and below to p with respect to the arrangement on the grid. That is, $\nu(p) = \{q \in \Omega_n | \mu_n(p,q) = 1\}$. □

A snake will be defined using the notion of neighborhood as a sequence of pebbles that consecutively lies in neighborhood of a pebble that proceeds.

Definition 6 ***(Snake).*** A *snake* s of size k is a sequence of pebbles $s = [s_1, s_2, \ldots, s_k]$ such that $\forall i \in \{1,2,\ldots,k\}\ s_i \in \Omega_n$ and $\forall j \in \{2,3,\ldots,k\}\ s_j \in \nu(s_{j-1})$. Pebble s_1 is called a *head* of the snake; pebble s_k is called a *tail* of the snake. □

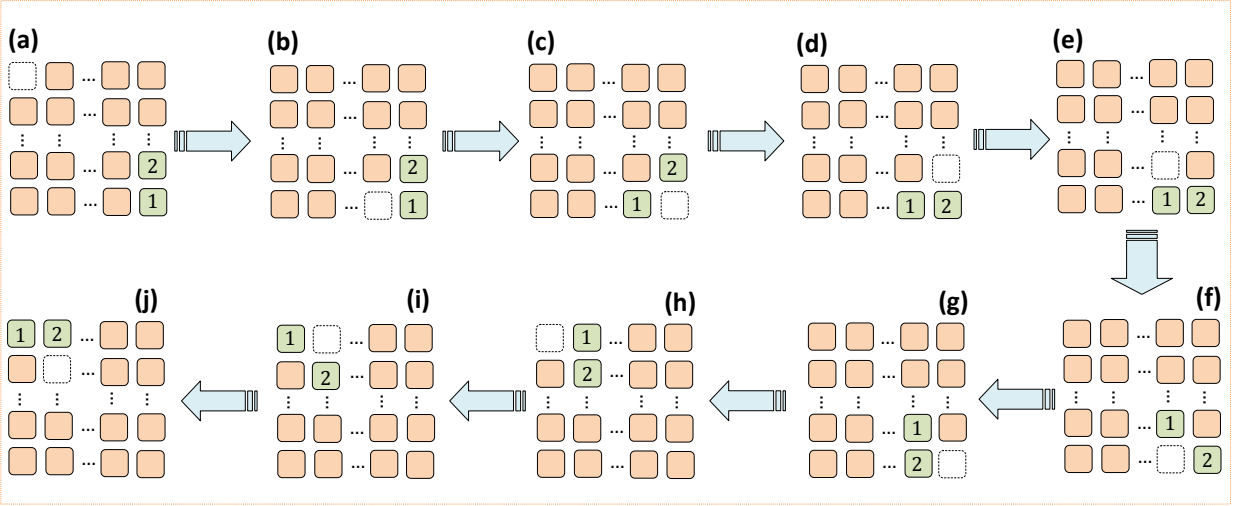


Figure 3. *Placing grouped pebbles using a snake.* The situation from Figure 2 is solved by grouping pebbles 1 and 2 into a snake, which is then moved as a whole from its original location in bottom right corner to the goal position in the upper left corner. The process consumes $12n + \mathcal{O}(1)$ which is approximately $4n$ better than the original approach that places pebbles individually.

Notice that each pebble itself forms a trivial snake of size 1. Composed movements of a snake *horizontally*, *vertically*, and *diagonally* can be defined analogically as in the case of a single pebble. If fact, they are generalizations of composed movements for single pebble. It is always assumed that the vacant position is in front of the head of snake in the direction of the intended movement. In such a setup, the snake can move forward by one position. The vacant position then needs to be moved around the snake in front of its head again to allow the next movement forward. See Figure 4 for illustration of composed movements for snakes (movements for a snake of length 2 are shown; it is easy to generalize composed movements for snakes of arbitrary length).

The horizontal and vertical composed movements consume $2k + 3$ moves. The number of moves consumed by the diagonal movement depends on the shape of a snake in the middle section – it is not that easy to express. However, if we need to move a snake of length 2 diagonally forward following the shape from Figure 4, then it consumes 10 moves.

Unfortunately it is rarely the case that a group of pebbles in some distant from goal location forms a snake. Even it is not that frequent that pebbles which are to be placed consecutively are close to each other. Hence, to take the advantage of moving a group of

pebbles as a snake we need first to form a snake of them. This is however not for free as a number of moves are necessary to form a snake. Thus, it is advisable to consider whether forming a snake is worthwhile. Moreover, there are many ways how to form a snake while each may be of different cost in terms of the number of moves.

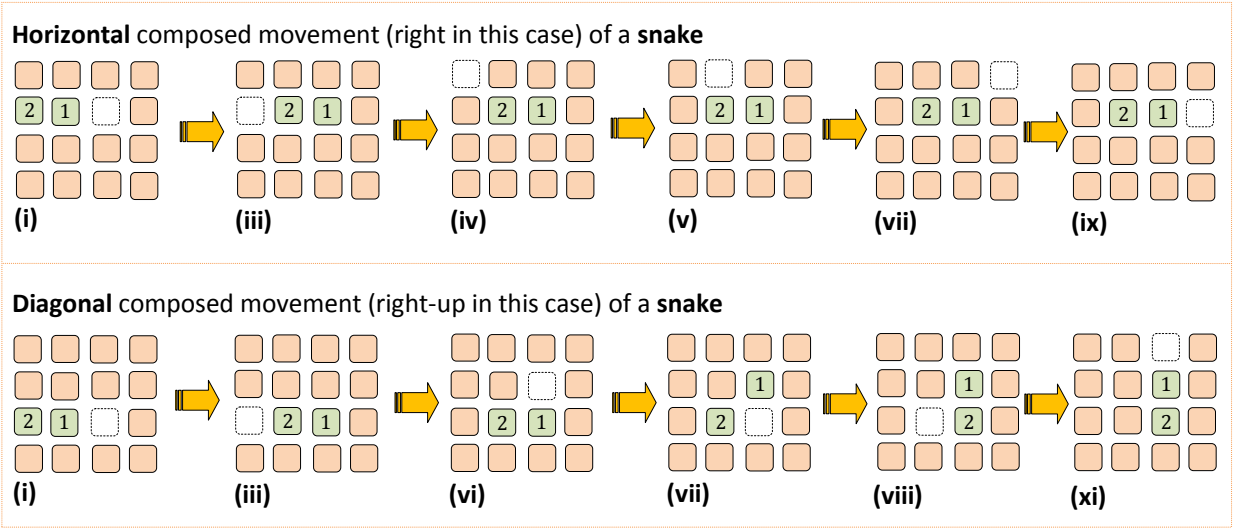


Figure 4. *Composed movements of a snake of length 2.* The horizontal and diagonal composed movements of a snake of length 2 are shown. Other cases as well as generalization for snakes of arbitrary length are straightforward.

Generally, the simplest way is to move one pebble to the other or vice versa in order to form a snake of length 2. It is known by using above calculations what number of moves is consumed by moving a snake as well as what number of moves are consumed by moving a pebble towards other pebble. Hence, it is easy to estimate the cost of using a snake in either of both ways as well as the cost of not using it at all in terms of the number of moves. Thus, it is possible to choose the most efficient option. This is another core idea of our new algorithm.

4.2. *A 'Snake' Based Algorithm*

Our new algorithm for solving $(n^2 - 1)$-puzzle will use snakes of length 2. The algorithm proceeds in the same way as the original algorithm of Parberry [4]. That is, pebbles are placed into the first row and then into the first column and after the first row and the first column are finished the task is reduced to the puzzle of the same type but smaller (namely, the task is reduced to solve $((n-1)^2 - 1)$-puzzle). The trivial case of 8-puzzle on a grid of the size 3×3 is again solved by the A* algorithm [7].

Along the solving process, the concept of snakes is used to move pebbles in a more efficient way. The basic idea is to make an estimation whether it will be beneficial to form a snake of two pebbles that are about to be placed. If so then a snake is formed in

one of the two ways – the first pebble is moved towards the second one or vice versa – the better option according to the estimations is always chosen. If forming a snake turns out not to be beneficial then pebbles are moved in the same way as in the case of the original algorithm; that is, one by one.

Algorithm 2. *The main function of a new algorithm for solving* (n^2-1)*-puzzle.* The function for producing a sequence of moves for placing two consecutive pebbles using snakes (if using snakes turns out to be beneficial) is shown.

function *Place-Pebbles*$(x_n, y_n, x_n^+, y_n^+, p, q)$: **pair**
  /* A function that produces a sequence of moves for placing two consecutive pebbles
  with respect to the order of placement. The new arrangement is returned in a return value.
  Parameters: x_n, x_y - a current arrangement of pebbles in the grid,
  x_n^+, y_n^+ - a goal arrangement of pebbles in the grid,
  p, q - two consecutive pebbles that will be placed. */
1: $c \leftarrow \text{cost}_1(x_n, y_n, x_n^+, y_n^+)(p, q)$
2: $e_{p,q} \leftarrow \text{estimate}_{\text{snake}}(x_n, y_n, x_n^+, y_n^+)(p, q)$
3: $e_{q,p} \leftarrow \text{estimate}_{\text{snake}}(x_n, y_n, x_n^+, y_n^+)(q, p)$
4: **if** $\min\{e_{p,q}, e_{q,p}\} < 1.2c$ **then**
5: **if** $e_{p,q} < e_{q,p}$ **then**
6: **let** (i, j) be a position such that $|i - x_n(p)| + |j - y_n(p)| = 1$
7: $(x_n, y_n) \leftarrow$ *Move-Vacant*(x_n, y_n, i, j)
8: $d_{min} \leftarrow \min\{|i' - x_n(p)| + |j' - y_n(p)|\,|\, i', j' \in \mathbb{N} \wedge |i' - x_n(q)| + |j' - y_n(q)| = 1\}$
9: **let** (i, j) be a position such that $|i - x_n(p)| + |j - y_n(p)| = d_{min}$
10: $(x_n, y_n) \leftarrow$ *Place-Pebble*(x_n, y_n, i, j, p)
11: **else**
12: **let** (i, j) be a position such that $|i - x_n(q)| + |j - y_n(q)| = 1$
13: $(x_n, y_n) \leftarrow$ *Move-Vacant*(x_n, y_n, i, j)
14: $d_{min} \leftarrow \min\{|i' - x_n(q)| + |j' - y_n(q)|\,|\, i', j' \in \mathbb{N} \wedge |i' - x_n(p)| + |j' - y_n(p)| = 1\}$
15: **let** (i, j) be a position such that $|i - x_n(q)| + |j - y_n(q)| = d_{min}$
16: $(x_n, y_n) \leftarrow$ *Place-Pebble*(x_n, y_n, i, j, q)
17: **let** $s = [p, q]$ be a snake {actually p and q form a snake at this point}
18: **let** π be a shortest path from $(x_n(p), y_n(p))$ to $(x_n^+(p), y_n^+(p))$ such that
  $\pi[|\pi| - 1] = (x_n^+(q), y_n^+(q))$ and π does not intersect any position
  containing already placed pebble
19: **for** $k = 1,2, \ldots, |\pi| - 1$ **do**
20: $(x_n, y_n) \leftarrow$ *Snake-Composed-Movement*$(x_n, y_n, \pi[k], \pi[k+1], s)$
  {when vacant position is moved it should avoid already placed pebbles}
21: **else**
22: $(x_n, y_n) \leftarrow$ *Place-Pebble*$(x_n, y_n, x_n^+, y_n^+, p)$
23: $(x_n, y_n) \leftarrow$ *Place-Pebble*$(x_n, y_n, x_n^+, y_n^+, q)$
24: **return** (x_n, y_n)

Let $\text{estimate}_{\text{snake}}(x_n, y_n, x_n^+, y_n^+): \Omega_n \times \Omega_n \longrightarrow \mathbb{N}_0$ is a functional that estimates the number of moves necessary to place a given two pebbles using the snake like motion. More precisely, $\text{estimate}_{\text{snake}}(x_n, y_n, x_n^+, y_n^+)(p, q)$ is the estimation of the number of

moves necessary to form a snake by moving pebble p towards q and to place the formed snake into the goal location where x_n, y_n and x_n^+, y_n^+ denote the current and the goal arrangements respectively. Notice, that $\text{estimate}_{\text{snake}}(x_n, y_n, x_n^+, y_n^+)$ can be calculated as sum of distances between several sections multiplied by number of moves needed to travel a unit of distance in that section. However, as different shapes of snake may occur, this calculation may not be exact. Next, let $\text{cost}_1(x_n, y_n, x_n^+, y_n^+): \Omega_n \times \Omega_n \longrightarrow \mathbb{N}_0$ be a functional that calculates exact number of moves necessary to place given two pebbles individually. As the case of individual pebbles is not distorted by any irregularities (such as different shapes as in the case of snake) the number of moves can be calculated exactly – again it is the sum of distances between given sections multiplied by the number of moves needed to travel unit distance in the individual sections.

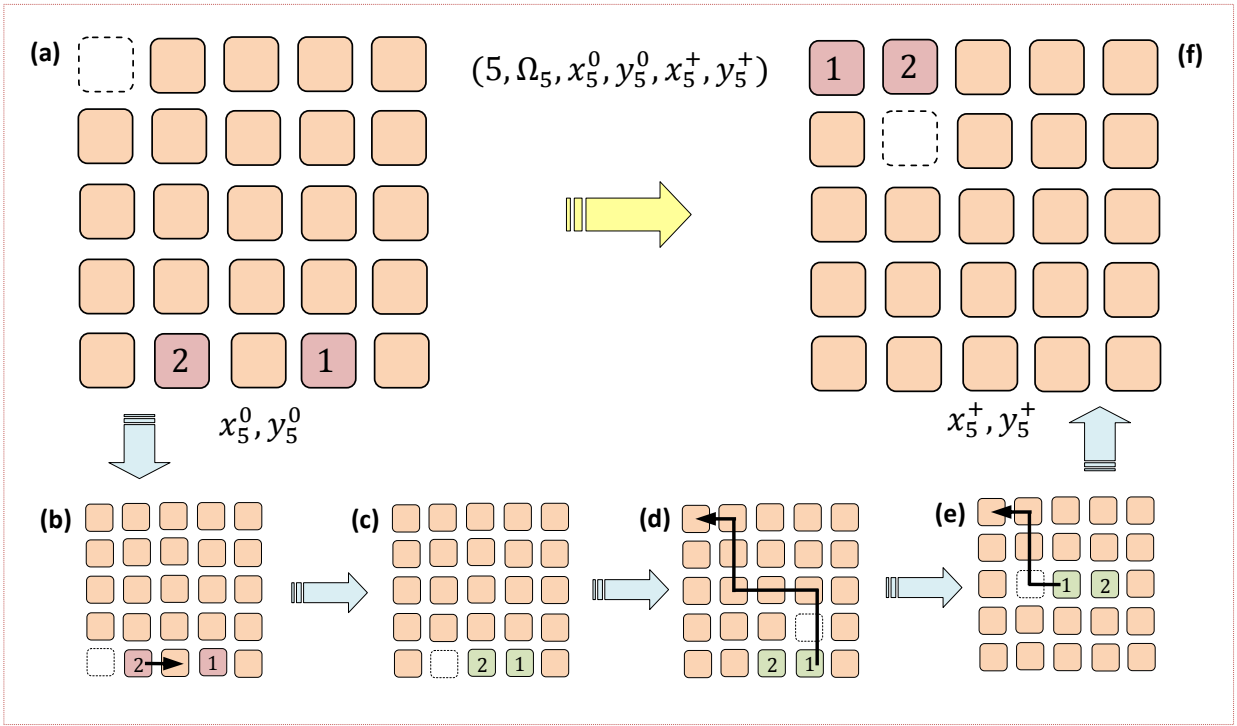


Figure 5. *Illustration of snake formation.* A snake will be formed by moving pebble 2 towards pebble 1 and then the whole snake will move to its goal location. The other way of forming a snake is to move pebble 1 towards pebble 2 and then to move the whole snake.

A preliminary experimental evaluation has shown that it suitable to use the following decision rule: if $\min\{\text{estimate}_{\text{snake}}(x_n, y_n, x_n^+, y_n^+)(p, q), \text{estimate}_{\text{snake}}(x_n, y_n, x_n^+, y_n^+)(p, q)\} < 1.2\text{cost}_1(x_n, y_n, x_n^+, y_n^+)(p, q)$ holds then it is tried to form a snake in the better of two ways and to compare the number of moves when snake is used with $\text{cost}_1(x_n, y_n, x_n^+, y_n^+)(p, q)$. If snake is still better then it is actually used to produce sequence of moves into the solution. Otherwise, the original way of placement of pebbles one by one is used.

The main function *Place-Pebbles* for placing a pair of pebbles using snake like motions is shown using pseudo-code as Algorithm 2. It is supposed that the function is used within the main loop of the solving algorithm (Algorithm 1). Several primitives, which all gets current arrangement of pebbles as its first two parameters, are used within Algorithm 2: a function *Move-Vacant* moves the vacant position to a specified new location; a function *Place-Pebble* implements the pebble placement process from the original algorithm of Parberry – here it is used as generic procedure to move pebble from one position to another. Finally, a *Snake-Composed-Movement* is a function that implements composed movements of a specified snake; two positions are specified – the current position of the head of snake and the new position for the head. It is also assumed that movement of the snake does not interfere with already placed pebbles. An example of snake formation and its placement is shown in Figure 5.

4.3. *Discussion on Longer Snakes*

We have also considered usage of snakes of length greater than 2. However, certain difficulties preclude using them effectively. There are many more options how to form a snake of length greater than 2. In the case of length k, there are at least $k!$ basic options how a snake can be formed (the order of pebbles is determined and then the snake collects pebbles in this order). Moreover, those do not include all the options (for example, it may be beneficial to form two snakes instead of a long one and so on). Therefore considering all the options and choosing the best one is computationally infeasible. Hence, using snakes of length 2 seems to be a good trade-off.

4.4. *Theoretical Analysis*

Although our new algorithm produces locally better sequence of moves for placing a pair of pebbles, it may not be necessarily better globally. Consider that different way of placing the pair of pebbles rearranges other pebbles differently as well, which may influence subsequent movements. Hence, theoretical analysis is quite difficult here. To evaluate the benefit of the new technique in a more realistic manner, we need some experimental evaluation. Nevertheless, theoretical analysis of worst cases can be done at least to get basic insight.

It has been shown that the original algorithm can always find a solution of the length at most $5n^3 + \frac{9}{2}n^2 + \frac{19}{2}n - 89$; that is, $5n^3 + \mathcal{O}(n^2)$ [4].

Proposition 1 (Worst-case Solution Length). Our new algorithm based on snakes can always produce a solution to a given instance of $(n^2 - 1)$-puzzle of the length of at most $\frac{14}{3}n^3 + \mathcal{O}(n^2)$. ■

Proof. It can be observed that the worst situation for the algorithm using snakes is when the two pebbles – let us denote them p and q – that are about to be placed are located in the last row or column. In such a case, we need $14n + \mathcal{O}(1)$ moves in the worst case.

Without loss of generality let us suppose both pebbles p and q to be placed in the last row while p is in the first column and q is in the last column. Exactly it is needed: at most $2n-1$ moves to move the vacant position near q; then at most $5(n-1)$ moves to move q towards p which forms a snake; and finally $7n+\mathcal{O}(1)$ moves to relocate the snake into the first row of the grid.

The algorithm needs to place $n-1$ pairs of pebbles and one pebble individually. Observe that moving one pebble individually to its goal position requires at most $8n$ moves. Hence, the first row and the first column requires at most $14n^2+c_1n+c_0$ moves where $c_0, c_1 \in \mathbb{R}$ with $c_0, c_1 \geq 0$. Let $M(n)$ denotes number of moves needed to solve the (n^2-1)-puzzle of size $n \times n$ then it holds that $M(n) \leq M(n-1)+14n^2+c_1n+c_0$. The solution of this inequality is $M(n)=\frac{14}{3}n^3+\mathcal{O}(n^2)$. ■

Notice, that this result does not show that our new algorithm is actually better than the original one. It merely shows that better theoretical estimation of the total number of moves can be done for it.

Proposition 2 (Worst-case Time Complexity). Our new algorithm based on snakes has the worst case time complexity of $\mathcal{O}(n^3)$. ■

Proof. The total time consumed by calls of *Move-Vacant* and *Place-Pebble* is linear in the number of moves that are performed. The time necessary to find shortest path avoiding already placed vertices is linear as well since the path has always a special shape that is known in advance (diagonal followed by horizontal or vertical). There is no need to use any path-search algorithm.

Time necessary for calculating $\text{estimate}_{\text{snake}}$ is at most the time necessary to finish the call of *Place-Pebble*, that is, linear in the number of moves again.

Finally, we need to observe that the call of *Snake-Composed-Movement* consumes time linear in the number of moves again since first the shortest path of the special shape needs to be found and then a snake needs to be moved along the path. ■

If theoretical results are summarized, we obtained a better upper bound for worst-case length of the solution for our new algorithm than it was obtained for the original one in [4]. It means that moving two pebbles together allows us to reduce worst-case estimation on the number of moves than if they are moved individually.

5. Experimental Evaluation

An experimental evaluation is necessary to explore qualities of our new snake-based algorithm comparatively with the algorithm of Parberry as we have only the upper bound estimation of the total number of steps so far which however does not show that the new algorithm actually produces fewer moves.

Algorithm based on snakes as well as that of Parberry were implemented in C++ to make experimental evaluation possible. A series of tests has been conducted to measure the total number of moves performed by each algorithm. The runtime necessary to solve the given instance has been measured too.

Measurements have been done for various sizes of the puzzle ranging from 3 to 50 (that is, n was ranging from 3 to 50). For each size of the puzzle, 40 solvable instances with random initial and goal arrangement of pebbles were generated (notice, that solvability can be detected by permutation parity check). Each generated instance was then solved by both algorithms and data was collected during the solving process.

The complete source and additional experimental data are provided at the website: http://ktiml.mff.cuni.cz/~surynek/research/j-puzzle-2011. This is to allow reproducibility of presented results.

5.1. *Competitive Comparison*

The competitive comparison of the total number of moves made by the snake-based algorithm and the algorithm of Parberry is shown in Figure 6. The improvement achieved by snake-based approach is illustrated as well. For each size of the instance, average out of 40 random instances is shown.

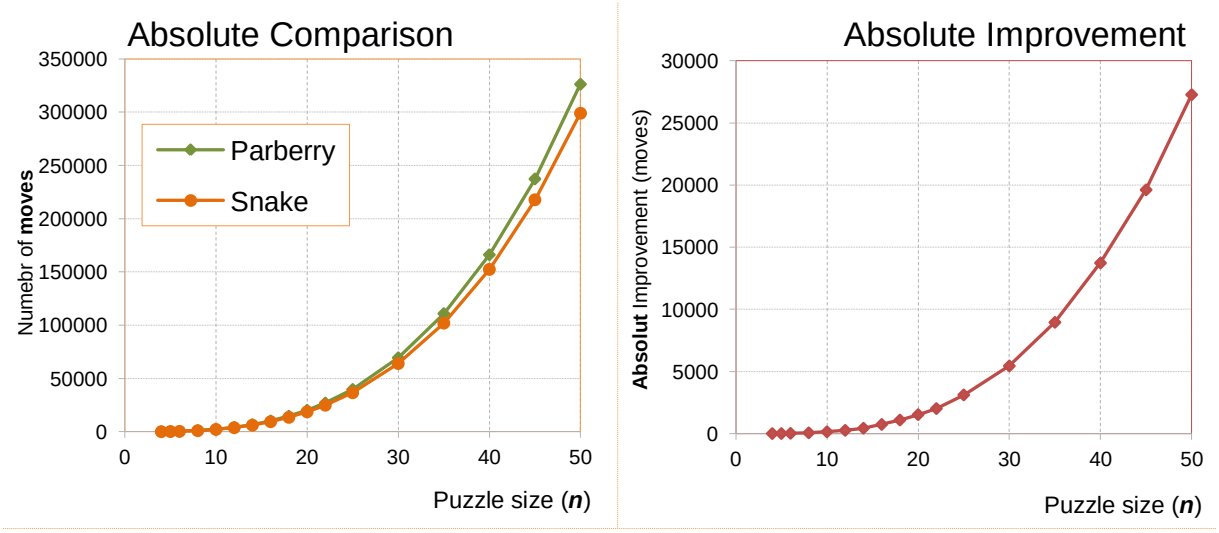


Figure 6. *Comparison of the original (Parberry) and snake based algorithm in terms of total number of steps.* Comparison has been done for several sizes of the puzzle ranging from 3 to 50. 40 random instances were generated for each size of the puzzle. The average number of moves for both algorithms is shown in the left part. The absolute improvement that can be achieved by using snakes is shown in the right part.

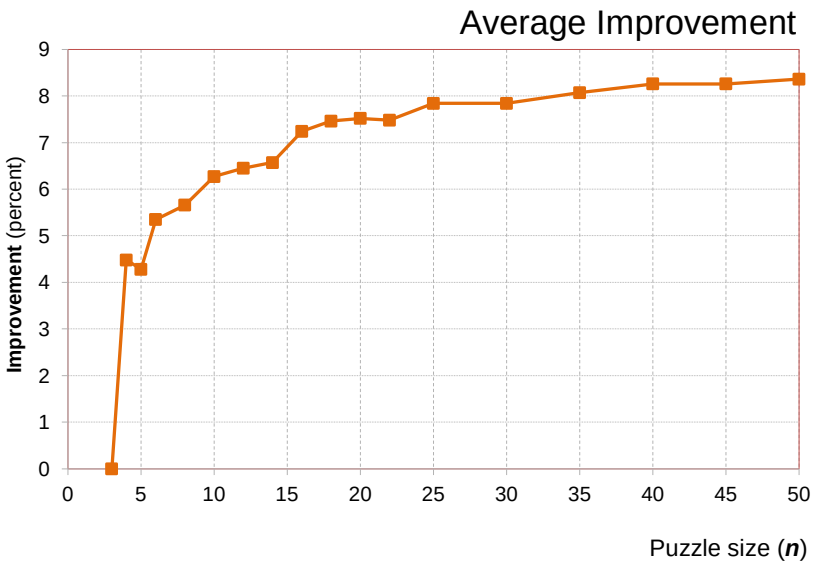
It is observable that the growth of the number of moves for growing size of the instance is polynomial indeed. Next, it can be observed that snake-based algorithm achieves a stable improvement, which is proportional to the total number of moves. The

more detailed insight into the achieved improvement of the total number of moves is provided by Figure 7. It clearly indicates that the improvement is becoming stable between 8% and 9% with respect to the original algorithm, as instances are getting larger.

Relative Improvement	
n	**Length Improvement (%)**
3	0.00
4	4.48
5	4.28
6	5.35
8	5.66
10	6.27
12	6.45
14	6.57
16	7.24
18	7.46
20	7.52
22	7.48
25	7.84
30	7.84
35	8.07
40	8.26
45	8.26
50	8.36

Table 1. *Relative improvement achieved by using snakes with respect to the original algorithm.* Again, the improvement has been measured for several sizes of the puzzle ranging from 3 to 50. For each size, 40 random instances were generated and the average improvement was calculated.

Figure 7. *Illustration of the trend in the average improvement.* It can be observed that the relative improvement tends to stabilize between 8% and 9% as instances are getting larger.



5.2. *Individual Instances*

Comparison of the total number of moves on the individual instances of various sizes is shown in Figure 8. These results show that using snakes, even though it is locally a better choice, can lead to global worsening of the solution. This phenomenon sometimes occurs exclusively on small instances. Here it is visible for instances of the size of 4×4. On larger instances, the local benefit of using snakes predominates over any local worsening of the arrangement so there is stably significant improvement of 7% and 9%. Notice, that this is not the average improvement calculated from several instances; this is improvement on a single instance.

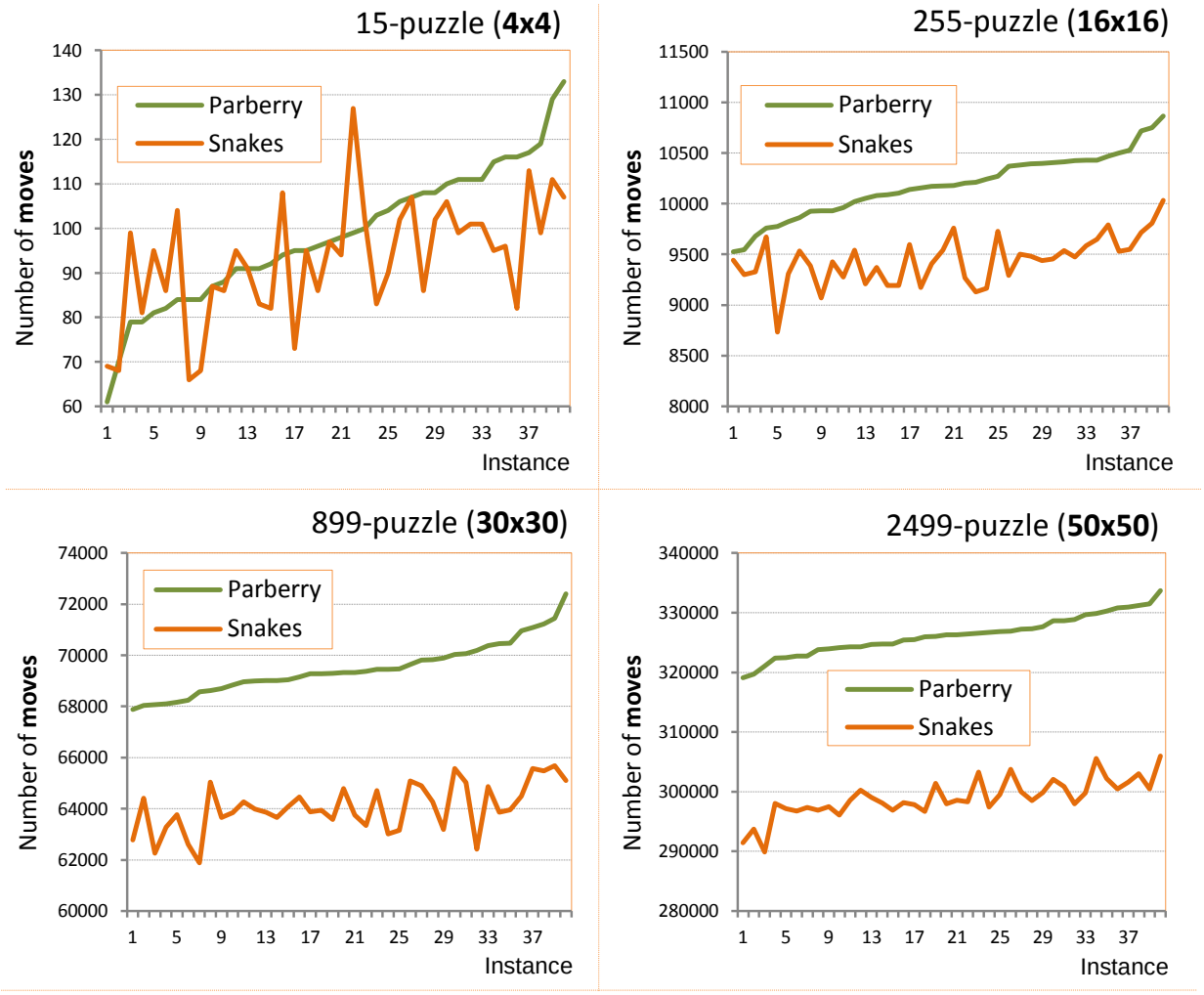


Figure 8. *Development of the improvement with the growing size of the puzzle instance.* Comparison of the number of moves conducted by the algorithm of Parberry and by our snake-based algorithm is shown for four puzzles of the increasing size. Individual instance for each size of the puzzle are sorted according to the increasing number of steps made by Parberry algorithm. It is observable that a worsening after applying snake-based approach may appear for small instances. The improvement is becoming stable (between 8-9%) for larger instances.

5.3. ***Runtime Measurement***

Finally, results regarding runtime are presented in Table 2. The average runtime for puzzles of size up to 50 × 50 are shown. Expectably, our algorithm based on snakes is slower as it makes decisions that are more complex (in fact, it is running the original algorithm plus snake placement to compare if snake is locally better). Nevertheless, the slowdown is acceptable.

Notice that both algorithms – Parberry and snake-based – are capable of solving puzzles with solutions consisting of hundreds of thousands of moves almost immediately. Hence, it can be concluded that both algorithms scales up extremely well and they can be used in on-line applications.

Table 2. *Runtime*[1] *measurements of our new algorithm and algorithm of Parberry.* Average time is calculated for each size of the puzzle out of 40 runs with different random setups. It can be observed that both algorithms scale up well.

	n	10	30	40	45	50
Time (seconds)	**Parberry**	< 0.10	< 0.10	< 0.10	0.10	0.10
	Snakes	< 0.10	< 0.10	< 0.10	0.10	0.19

5.4. *Summary of Experimental Evaluation*

The conducted experimental evaluation clearly shows that our new algorithm based on snakes outperforms the original algorithm of Parberry in terms of the quality of generated solutions (defined as total number of moves). Experiments support the claim that using snakes greedily, that is, if they are locally better, leads to global improvement of solution even though the current arrangement may be worsened sometimes from the global point of view. As instances are getting larger, the improvement tends to stabilize itself between 8% and 9% in average. Even on larger instances – that is larger than 30×30 – possible fluctuations towards worsening the solution are eliminated, hence using snakes expectably leads to an improvement of 7% to 9% on an individual instance.

Runtime measurements show that both tested algorithms solve instances of tested sizes in less than $0.2s$. Thus, it can be concluded that scalability is extremely good.

6. Conclusions and Future Work

We have presented a new polynomial-time algorithm for solving $(n^2 - 1)$-puzzle in an on-line mode sub-optimally. The algorithm is based on an idea to move pebbles jointly in groups called snakes, which was supposed to reduce the total number of moves. The experimental evaluation eventually confirmed this claim and showed that the new algorithm outperforms the existent state-of-the-art algorithm of Parberry [4] by 8% to 9% in terms of the average length of the solution. Theoretical upper bounds on the length of the solution are also better for the new algorithm as we have shown. Regarding runtime the new algorithm is marginally slower due to its more complex computations, however this is absolutely acceptable for any real-life application as the runtime is linear in the number of produced moves (approximately 10^6 moves can be produced per second).

It will be interesting for future work to add more measures for reducing the total number of moves towards the optimum. Observe that choosing a more promising local rearrangement among several options can be easily parallelized.

[1] All the tests were run on a commodity PC with CPU Intel Core2 Duo 3.00 GHz and 2 GiB of RAM under Windows XP 32-bit edition. The C++ code was compiled with Microsoft Visual Studio 2008 C++ compiler.

We are also interested in generalized variants of (n^2-1)-puzzle where there is more than one vacant position. These variants are known as (n^2-k)-puzzle with $k>1$ [11]. Although it seems that obtaining optimal solutions remains hard in this case, multiple vacant positions can be used to rearrange pebbles more efficiently in the sub-optimal approach.

Finally, it is interesting for us to study techniques for optimal solving of this and related problems; especially the case with small unoccupied space (that is, with $k \ll n^2$). This is quite open area as today's optimal solving techniques [13] can manage only small number of pebbles compared to the size of the unoccupied space.

Acknowledgments

This work is partially supported by The Czech Science Foundation (Grantová agentura České republiky - GAČR) under the contract number 201/09/P318 and by The Ministry of Education, Youth and Sports, Czech Republic (Ministerstvo školství, mládeže a tělovýchovy ČR – MŠMT ČR) under the contract number MSM 0021620838. It is also partially supported by the Japan Society for the Promotion of Science (JSPS) within the post-doctoral fellowship of the first author (reference number P11743).

References

1. **Kornhauser**, D., **Miller**, G. L., and **Spirakis**, P. G.: *Coordinating Pebble Motion on Graphs, the Diameter of Permutation Groups, and Applications.* Proceedings of the 25th Annual Symposium on Foundations of Computer Science (FOCS 1984), pp. 241-250, IEEE Press, 1984.
2. **Luna**, R., **Bekris**, K. E.: *Push and Swap: Fast Cooperative Path-Finding with Completeness Guarantees.* Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011), pp. 294-300, IJCAI/AAAI, 2011.
3. **Michalík**, P.: *Sub-optimal Algorithms for Solving Sliding Puzzles.* Master thesis, Charles University in Prague, Czech Republic, 2011.
4. **Parberry**, I.: *A real-time algorithm for the (nˆ2-1)-puzzle*. Information Processing Letters, Volume 56 (1), pp. 23-28, Elsevier, 1995.
5. **Ratner**, D. and **Warmuth**, M. K.: *Finding a Shortest Solution for the N×N Extension of the 15 PUZZLE Is Intractable.* Proceedings of the 5th National Conference on Artificial Intelligence (AAAI 1986), pp. 168-172, Morgan Kaufmann Publishers, 1986.
6. **Ratner**, D. and **Warmuth**, M. K.: *NxN Puzzle and Related Relocation Problems.* Journal of Symbolic Computation, Volume 10 (2), pp. 111-138, Elsevier, 1990.
7. **Russel**, S., **Norvig**, P.: *Artificial Intelligence – A modern approach.* Prentice Hall, 2003.
8. **Ryan**, M. R. K.: *Graph Decomposition for Efficient Multi-Robot Path Planning.* Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007), pp. 2003-2008, IJCAI, 2007.
9. **Ryan**, M. R. K.: *Exploiting Subgraph Structure in Multi-Robot Path Planning.* Journal of Artificial Intelligence Research (JAIR), Volume 31, 2008, pp. 497-542, AAAI Press, 2008.
10. **Surynek**, P.: *A Novel Approach to Path Planning for Multiple Robots in Bi-connected Graphs.* Proceedings of the 2009 IEEE International Conference on Robotics and Automation (ICRA 2009), pp. 3613-3619, IEEE Press, 2009.
11. **Surynek**, P.: *An Application of Pebble Motion on Graphs to Abstract Multi-robot Path Planning.* Proceedings of the 21st International Conference on Tools with Artificial Intelligence (ICTAI 2009), pp. 151-158, IEEE Press, 2009.

12. **Surynek**, P.: *An Optimization Variant of Multi-Robot Path Planning is Intractable.* Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI 2010), pp. 1261-1263, AAAI Press, 2010.
13. **Standley**, T.: *Finding Optimal Solutions to Cooperative Pathfinding Problems.* Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI-2010), pp. 173-178, AAAI Press, 2010.
14. **Wang**, K. C., **Botea**, A.: Scalable Multi-Agent Pathfinding on Grid Maps with Tractability and Completeness Guarantees. Proceedings of ECAI 2010 - 19th European Conference on Artificial Intelligence (ECAI 2010), pp. 977-978, Frontiers in Artificial Intelligence and Applications 215, IOS Press, 2010.
15. **Wang**, K. C., **Botea**, A., **Kilby**, P.: *Solution Quality Improvements for Massively Multi-Agent Pathfinding.* Proceedings of the 25th AAAI Conference on Artificial Intelligence (AAAI 2011), AAAI Press 2011.
16. **West**, D. B.: *Introduction to Graph Theory.* Prentice Hall, 2000.
17. **Westbrook**, J., **Tarjan**, R. E.: *Maintaining bridge-connected and bi-connected components on-line.* Algorithmica, Volume 7, Number 5&6, pp. 433–464, Springer, 1992.
18. **Wilson**, R. M.: *Graph Puzzles, Homotopy, and the Alternating Group.* Journal of Combinatorial Theory, Ser. B 16, pp. 86-96, Elsevier, 1974.