CentraleSupélec

Discrete Optimization

Peg Solitaire solver

Authors :
M. Ahmed El Aamrani
M. Wassim Lhourti

Professor :
M. Hugues Talbot

June 7^{th} 2018

Contents

Introduction

Peg solitaire is a one player game using pegs and a board with some holes. In any given configuration, each hole contains at most one peg (see figure 1 : a dark circle implies a hole with a peg, a blank circle implies a hole with no peg.). A peg solitaire game has two special configurations: the starting configuration and the finishing configuration. The aim of this game is to get the finishing configuration from the starting configuration by moving and removing pegs.

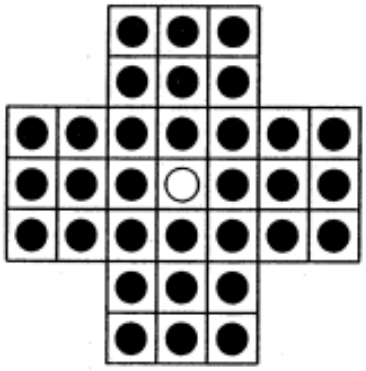


Figure 1: Starting configuration example

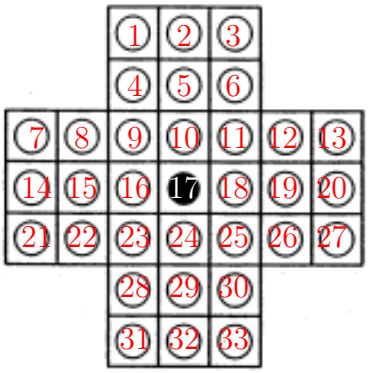


Figure 2: Finishing configuration example and holes indexing

Chapter 1

Integer programming

1.1 Description

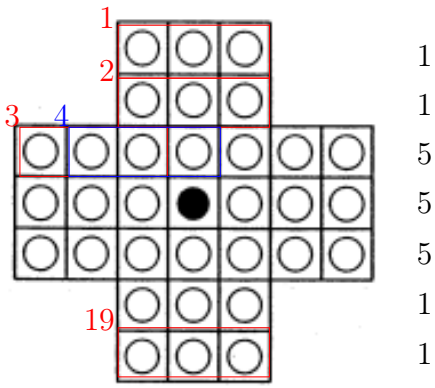
This section has been extracted and detailed from an original paper [3]

We assume that all the holes on a given board are indexed by integer numbers $\{1, 2, \ldots, n\}$. The board of Figure 1 has 33 holes, which implies $n = 33$. We describe a state of certain configuration (pegs in the holes) by a binary vector $\boldsymbol{p}$ of length n satisfying that the i-th element of $\boldsymbol{p}$ is 1 if and only if the hole i contains a peg. We can take for example the order of i while enumerating holes horizontally (see Figure 2). We denote the starting configuration by $\boldsymbol{p_s}$ and the finishing configuration by $\boldsymbol{p_f}$. Therefore , we have:

$$\overset{17}{\boldsymbol{p_s} = [1 \ldots 1 \ 0 \ 1 \ldots 1]}$$
$$\boldsymbol{p_f} = [0 \ldots 0 \ 1 \ 0 \ldots 0]$$

Let J be the family of all the sequences of consecutive three holes on a given board. Each element in J corresponds to a certain jump and so we can denote a jump by a unit vector indexed by J. In the rest of this paper, we assume that all the elements in J are indexed by $\{1, 2, , \ldots, m\}$. For example, the board of Figure 1 contains 76 sequences of consecutive three holes, which means $m = 76$. For example, Figure 1.1 depicts the 19 horizontal jumps.

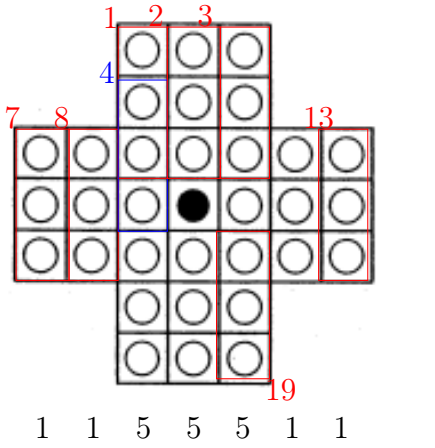
We continue indexing following this same logic. For instance, the left-most jump in the 4-th row has an index *8*, and so on ... These horizontal jumps can be either to the right or to the left. We can choose for example, that J contains first the 19 jumps to the right with the previous order, followed by these 19 jumps to the left. Jumps to the right refer to the jumps that fill a hole by moving to the right. Jumps to the left refer to the jumps that fill a hole by moving to the left.



$4 * 1 + 3 * 5 = 19$

Figure 1.1: Horizontal jump indexing

We can do the same with vertical jumps as follows :



$4 * 1 + 3 * 5 = 19$

Figure 1.2: Vertical jump indexing

These vertical jumps can be either to the top or down. Jumps to the top refer to the jumps that fill a hole by moving to the top. Jumps down refer to the jumps that fill a hole by moving down. For the set J, we can choose the following indexing :

$$\text{J} = [19 \rightarrow \text{jumps}, \quad 19 \leftarrow \text{jumps}, \quad 19 \uparrow \text{jumps}, \quad 19 \downarrow \text{jumps}]$$

Each of the 4 subsets follow the indexing which was defined previously. Furthermore, given a peg solitaire board, we define a n x m matrix $A = (a_{ij})$ whose rows and columns are indexed by holes and jumps respectively with :

$$a_{ij} = \begin{cases} 1 & \text{(a peg on the hole i is removed by the jump j)} \\ -1 & \text{(a peg is placed on the hole i by the jump j)} \\ 0 & \text{(otherwise)} \end{cases} \tag{1.1}$$

In this case, we can represent A as the following:

$$A = \left[\; A_1 \; \middle| \; A_2 \; \middle| \; A_3 \; \middle| \; A_4 \; \right] \qquad \text{where } A_i \text{ is a 33 x 19 matrix}$$

Following the previous defined indexing, we have :

$$A_1 = \begin{pmatrix}
1 \\
1 \\
-1 \\
& 1 \\
& 1 \\
& -1 \\
& & 1 \\
& & 1 & 1 \\
& & -1 & 1 & 1 \\
& & & -1 & 1 & 1 \\
& & & & -1 & 1 & 1 \\
& & & & & -1 & 1 \\
& & & & & & -1 \\
& & & & & & & 1 \\
& & & & & & & 1 & 1 \\
& & & & & & & -1 & 1 & 1 \\
& & & & & & & & -1 & 1 & 1 \\
& & & & & & & & & -1 & 1 & 1 \\
& & & & & & & & & & -1 & 1 \\
& & & & & & & & & & & -1 \\
& & & & & & & & & & & & 1 \\
& & & & & & & & & & & & 1 & 1 \\
& & & & & & & & & & & & -1 & 1 & 1 \\
& & & & & & & & & & & & & -1 & 1 & 1 \\
& & & & & & & & & & & & & & -1 & 1 & 1 \\
& & & & & & & & & & & & & & & -1 & 1 \\
& & & & & & & & & & & & & & & & -1 \\
& & & & & & & & & & & & & & & & & 1 \\
& & & & & & & & & & & & & & & & & 1 \\
& & & & & & & & & & & & & & & & & -1 \\
& & & & & & & & & & & & & & & & & & 1 \\
& & & & & & & & & & & & & & & & & & 1 \\
& & & & & & & & & & & & & & & & & & -1
\end{pmatrix}$$

$$A_2 = \begin{pmatrix}
-1 \\
1 \\
1 \\
& -1 \\
& 1 \\
& 1 \\
& & -1 \\
& & 1 & -1 \\
& & 1 & 1 & -1 \\
& & & 1 & 1 & -1 \\
& & & & 1 & 1 & -1 \\
& & & & & 1 & 1 \\
& & & & & & 1 \\
& & & & & & & -1 \\
& & & & & & & 1 & -1 \\
& & & & & & & 1 & 1 & -1 \\
& & & & & & & & 1 & 1 & -1 \\
& & & & & & & & & 1 & 1 & -1 \\
& & & & & & & & & & 1 & 1 \\
& & & & & & & & & & & 1 \\
& & & & & & & & & & & & -1 \\
& & & & & & & & & & & & 1 & -1 \\
& & & & & & & & & & & & 1 & 1 & -1 \\
& & & & & & & & & & & & & 1 & 1 & -1 \\
& & & & & & & & & & & & & & 1 & 1 & -1 \\
& & & & & & & & & & & & & & & 1 & 1 \\
& & & & & & & & & & & & & & & & 1 \\
& & & & & & & & & & & & & & & & & -1 \\
& & & & & & & & & & & & & & & & & 1 \\
& & & & & & & & & & & & & & & & & 1 \\
& & & & & & & & & & & & & & & & & & -1 \\
& & & & & & & & & & & & & & & & & & 1 \\
& & & & & & & & & & & & & & & & & & 1
\end{pmatrix}$$

$$A_3 = \begin{pmatrix}
-1&&&&&&&&&&&&&&&&&& \\
&-1&&&&&&&&&&&&&&&&& \\
&&-1&&&&&&&&&&&&&&&& \\
1&&&-1&&&&&&&&&&&&&&& \\
&1&&&-1&&&&&&&&&&&&&& \\
&&1&&&-1&&&&&&&&&&&&& \\
&&&&&&-1&&&&&&&&&&&& \\
&&&&&&&-1&&&&&&&&&&& \\
1&&&1&&&&&-1&&&&&&&&&& \\
&1&&&1&&&&&-1&&&&&&&&& \\
&&1&&&1&&&&&-1&&&&&&&& \\
&&&&&&&&&&&-1&&&&&&& \\
&&&&&&&&&&&&-1&&&&&& \\
&&&&&&1&&&&&&&&&&&& \\
&&&&&&&1&&&&&&&&&&& \\
&&&1&&&&&1&&&&&-1&&&&& \\
&&&&1&&&&&1&&&&&-1&&&& \\
&&&&&1&&&&&1&&&&&-1&&& \\
&&&&&&&&&&&1&&&&&&& \\
&&&&&&&&&&&&1&&&&&& \\
&&&&&&1&&&&&&&&&&&& \\
&&&&&&&1&&&&&&&&&&& \\
&&&&&&&&1&&&&&1&&&-1&& \\
&&&&&&&&&1&&&&&1&&&-1& \\
&&&&&&&&&&1&&&&&1&&&-1 \\
&&&&&&&&&&&1&&&&&&& \\
&&&&&&&&&&&&1&&&&&& \\
&&&&&&&&&&&&&1&&&1&& \\
&&&&&&&&&&&&&&1&&&1& \\
&&&&&&&&&&&&&&&1&&&1 \\
&&&&&&&&&&&&&&&&1&& \\
&&&&&&&&&&&&&&&&&1& \\
&&&&&&&&&&&&&&&&&&1
\end{pmatrix}$$

```
$$
a_{ij} = \left\{
   \begin{array}{ll}
      1 & \mbox{if} &\text{ (a peg on the hole i is removed by the jump j)} \\
      -1 & \mbox{if} & \text{ (a peg is placed on the hole i by the jump j)}\\
      0 &  \mbox{otherwise}&
   \end{array}
\right.
$$
```

$$A_4 = \begin{pmatrix}
1&&&&&&&&&&&&&&&&&& \\
&1&&&&&&&&&&&&&&&&& \\
&&1&&&&&&&&&&&&&&&& \\
1&&&1&&&&&&&&&&&&&&& \\
&1&&&1&&&&&&&&&&&&&& \\
&&1&&&1&&&&&&&&&&&&& \\
&&&&&&1&&&&&&&&&&&& \\
&&&&&&&1&&&&&&&&&&& \\
-1&&&1&&&&&1&&&&&&&&&& \\
&-1&&&1&&&&&1&&&&&&&&& \\
&&-1&&&1&&&&&1&&&&&&&& \\
&&&&&&&&&&&1&&&&&&& \\
&&&&&&&&&&&&1&&&&&& \\
&&&&&&1&&&&&&&&&&&& \\
&&&&&&&1&&&&&&&&&&& \\
&&&-1&&&&&1&&&&&1&&&&& \\
&&&&-1&&&&&1&&&&&1&&&& \\
&&&&&-1&&&&&1&&&&&1&&& \\
&&&&&&&&&&&1&&&&&&& \\
&&&&&&&&&&&&1&&&&&& \\
&&&&&&-1&&&&&&&&&&&& \\
&&&&&&&-1&&&&&&&&&&& \\
&&&&&&&&-1&&&&&1&&&1&& \\
&&&&&&&&&-1&&&&&1&&&1& \\
&&&&&&&&&&-1&&&&&1&&&1 \\
&&&&&&&&&&&-1&&&&&&& \\
&&&&&&&&&&&&-1&&&&&& \\
&&&&&&&&&&&&&-1&&&1&& \\
&&&&&&&&&&&&&&-1&&&1& \\
&&&&&&&&&&&&&&&-1&&&1 \\
&&&&&&&&&&&&&&&&-1&& \\
&&&&&&&&&&&&&&&&&-1& \\
&&&&&&&&&&&&&&&&&&-1
\end{pmatrix}$$

For any binary vector $\boldsymbol{p}$, $\#\boldsymbol{p}$ denotes the number of 1-s in the elements of $\boldsymbol{p}$. We denote $\#\boldsymbol{p_s} - \#\boldsymbol{p_f}$ by l. If the given peg solitaire problem is feasible, any feasible sequence consists of l jumps, because at each jump, the number of pegs decreases by 1. For example, the peg solitaire problem defined by Figures 1 and 2 is feasible and there exists a feasible sequence whose length is $l = 32$.

Since each jump corresponds to a unit vector of length m, a feasible sequence corresponds to a sequence of l unit vectors $(\boldsymbol{x}^1, \boldsymbol{x}^2, \ldots, \boldsymbol{x}^l)$ such that $\boldsymbol{x}^k = (x_1^k, x_2^k, \ldots x_m^k)$ for all $k \in \{1, 2, \ldots, l\}$ and :

$$x_j^k = \begin{cases} 1 & \text{the k-th move is the jump j} \\ 0 & \text{the k-th move is not the jump j} \end{cases} \tag{1.2}$$

If a configuration $\boldsymbol{p'}$ is obtained by applying the jump j to a configuration $\boldsymbol{p}$ then we have: $\boldsymbol{p'} = \boldsymbol{p} - A\boldsymbol{u}$, where $\boldsymbol{u}$ is the j-th unit vector in $\{0,1\}^m$. For instance, let us consider the following jump :

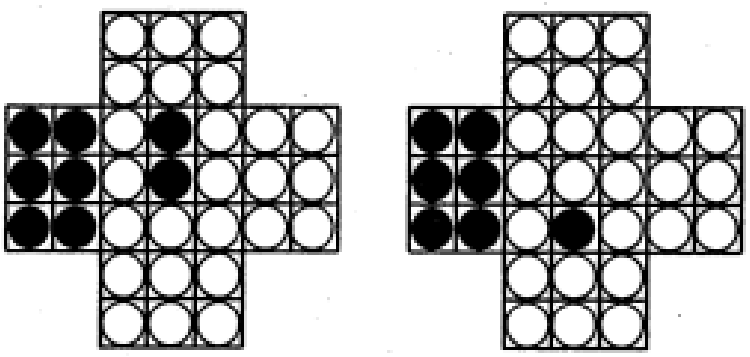


Figure 1.3: An example of a jump

In this case, we have :

$$\boldsymbol{p} = \left(0\,0\,0 \;\middle|\; 0\,0\,0 \;\middle|\; 1\,1\,0\,1\,0\,0\,0 \;\middle|\; 1\,1\,0\,1\,0\,0\,0 \;\middle|\; 1\,1\,0\,0\,0\,0\,0 \;\middle|\; 0\,0\,0 \;\middle|\; 0\,0\,0\right)^T$$
$$\boldsymbol{p'} = \left(0\,0\,0 \;\middle|\; 0\,0\,0 \;\middle|\; 1\,1\,0\,0\,0\,0\,0 \;\middle|\; 1\,1\,0\,0\,0\,0\,0 \;\middle|\; 1\,1\,0\,1\,0\,0\,0 \;\middle|\; 0\,0\,0 \;\middle|\; 0\,0\,0\right)^T$$

Furthermore, it corresponds to the (19x3+10 = 67)-th jump because it is a vertical jump down. Therefore Au_j can be assimilated to the 10-th column of A_4. Thus, we have:

$$\begin{aligned} \boldsymbol{p'} = &\left(0\,0\,0 \;\middle|\; 0\,0\,0 \;\middle|\; 1\,1\,0\,1\,0\,0\,0 \;\middle|\; 1\,1\,0\,1\,0\,0\,0 \;\middle|\; 1\,1\,0\,0\,0\,0\,0 \;\middle|\; 0\,0\,0 \;\middle|\; 0\,0\,0\right)^T \\ - &\left(0\,0\,0 \;\middle|\; 0\,0\,0 \;\middle|\; 0\,0\,0\,1\,0\,0\,0 \;\middle|\; 0\,0\,0\,1\,0\,0\,0 \;\middle|\; 0\,0\,0\,-1\,0\,0\,0 \;\middle|\; 0\,0\,0 \;\middle|\; 0\,0\,0\right)^T \\ = &\left(0\,0\,0 \;\middle|\; 0\,0\,0 \;\middle|\; 1\,1\,0\,0\,0\,0\,0 \;\middle|\; 1\,1\,0\,0\,0\,0\,0 \;\middle|\; 1\,1\,0\,1\,0\,0\,0 \;\middle|\; 0\,0\,0 \;\middle|\; 0\,0\,0\right)^T \end{aligned}$$

The statement is true in this special case, and is generalized for all jumps. Thus, we can formulate the peg solitaire problem as the following integer programming problem:

Peg Solitaire solver - Integer Program :

$$\text{Find } (\boldsymbol{x}^1, \boldsymbol{x}^2, \ldots, \boldsymbol{x}^l) \tag{1.3}$$
$$\text{s.t. } A(\boldsymbol{x}^1 + \boldsymbol{x}^2 + \ldots + \boldsymbol{x}^l) = \boldsymbol{p_s} - \boldsymbol{p_f} \tag{1.4}$$
$$\forall k \in \{1, 2, \ldots, l\} \;\; \mathbf{0} \leq \boldsymbol{p_s} - A(\boldsymbol{x}^1 + \boldsymbol{x}^2 + \ldots + \boldsymbol{x}^k) \leq \mathbf{1} \tag{1.5}$$
$$\forall k \in \{1, 2, \ldots, l\} \;\; x_1^k + x_2^k + \ldots + x_m^k = 1 \tag{1.6}$$
$$\forall k \in \{1, 2, \ldots, l\} \;\; \boldsymbol{x}^k \in \{0,1\}^m \tag{1.7}$$

1.2 Discussion

- We are looking only for a feasible solution (3), so we can take a zero matrix cost. It means that we do not select solutions upon a criteria, but we are only looking if these solutions exist, and if it is the case, show this solution to solve the game.

- We have $\boldsymbol{p'} = \boldsymbol{p} - A\boldsymbol{u}$. Thus, by induction we have (4). This constraint forces the intermediate configurations $\boldsymbol{p'}$ to have integer values.

- Nevertheless, these intermediate configurations should also correspond to a real peg configurations: meaning that they have zeros where there is no peg , and ones otherwise. Therefore, $\boldsymbol{p'} = \boldsymbol{p_s} - A(\boldsymbol{x}^1 + \boldsymbol{x}^2 + \ldots + \boldsymbol{x}^k)$ for $k \in \{1, 2, \ldots, l\}$ which represent all intermediate configurations should verify (5) then.

- A sub-sequence $\boldsymbol{x}^k$ with $k \in \{1, 2, \ldots, l\}$ from the whole sequence should correspond only to one jump from the m possible jumps. Therefore, we have conditions (6) and (7).

1.3 Implementation and validation

In order to implement this problem, we first concatenated the sequence of unknown vectors, into a single one. In this concatenation, we join jumps one after the other. It means that our unknown vector become :

$$\boldsymbol{x} = [x_1^1, x_2^1, \ldots, x_m^1 \quad \ldots\ldots \quad x_1^l, x_2^l, \ldots, x_m^l]$$

with $m = 76$ and $l = \#\boldsymbol{p}_s - \#\boldsymbol{p}_f$

Then, we've used `cvxopt` in Python (please refer to the Jupyter Notebook Code). We took a zero cost matrix, and implemented our equality and inequality matrices, and set all variables to binary variables.

To verify correctness of our matrices, we've looked for an example of a feasible solution for the classic English peg solitaire problem (see Figure 1 and 2). This feasible solution was available here[1]. Then, we translated first this feasible solution in a sequence of length 31 following our notations. Indeed, we have :

Sequence index	1	2	3	4	5	6	7	8	9	10
Jump index	61	3	57	41	19	57	51	12	32	63

Sequence index	11	12	13	14	15	16	17	18	19	20
Jump index	12	34	56	72	18	56	62	25	5	50

Sequence index	21	22	23	24	25	26	27	28	29	30	31
Jump index	25	15	49	24	22	64	13	27	52	28	8

[1]How To Solve Peg Solitaire, Yoolan, https://www.youtube.com/watch?v=Bt6GpGvUNeQ

After reshaping the feasible solution into a vector *fsol*, we've then verified that ($E * fsol == \boldsymbol{e}$) and ($I * fsol <= \boldsymbol{i}$) are both true, which is comforting, and can be considered as a validation for the definition of our matrices. *Please refer to 'In [6]' in the Jupyter Notebook code.*

Difficulty

- If we consider the classic English peg solitaire problem, with the starting and finishing configurations as in the Figure 1. and 2. Then, we have $l \times m = 31 \times 76 = 2356$ variables. Furthermore, we have $n + l = 33 + 31 = 64$ equality constraints and $2 \times l \times n = 2 \times 31 \times 33 = 2046$ inequality constraints.

- Since there are 2356 variables in our model, we can say that there exists $2^{2356} \approx 10^{3 \times 235} \approx 10^{700}$ feasible peg configuration !

- Therefore, the size of the integer programming problem is huge and so it is hard to solve the problem by personal computers. Using *cvxopt* with our personal computers, the algorithm kept searching for the solution during 5 hours before we interrupted it.

Hence, the first idea that we decided to explore is the use of a backtracking search approach that would help us solve our problem faster.

Chapter 2

Backtracking search

2.1 General algorithm

One way to solve the peg solitaire problem is to implement a backtrack search algorithm. From given start and finishing configurations, we make a *legal* jump. Then, we consider the newly obtained configuration as the new start position, and we keep recursively searching for a solution until we arrive to the finishing configuration, if we are dealing with a feasible problem. Otherwise, the algorithm returns that the problem is unfeasible.

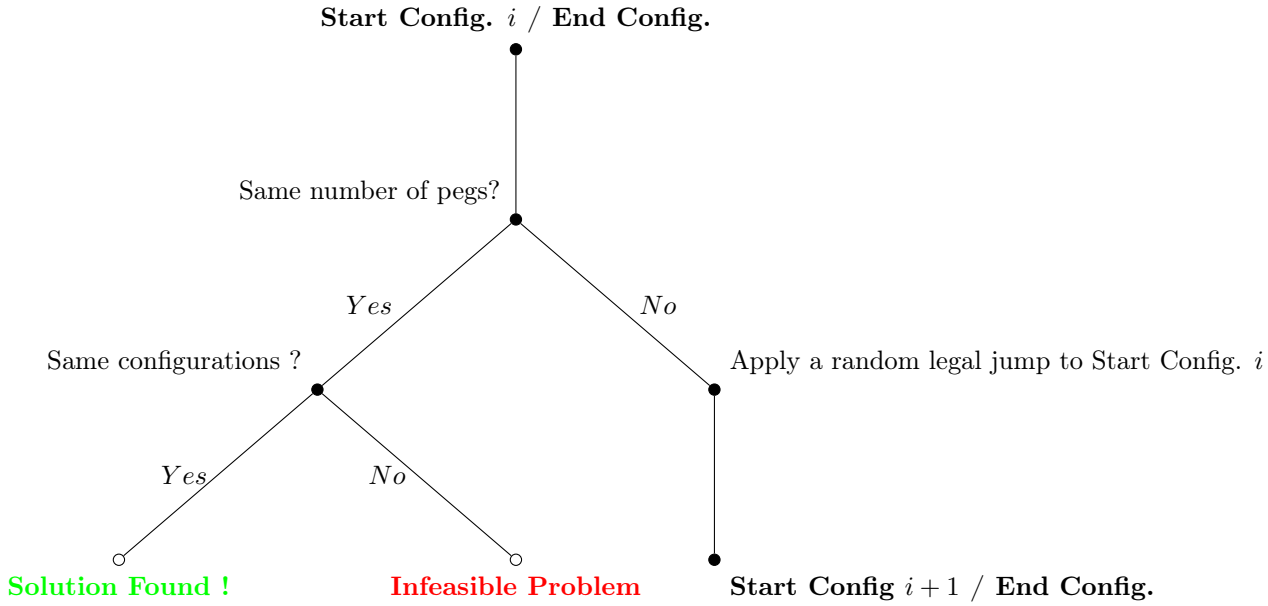


Figure 2.1: Backtrack search illustration

Furthermore, we decided to implement various pruning techniques in order to improve our algorithm's efficiency. In this project, we used simultaneously two techniques which provided us with useful ways to prune the search space. These methods are:

- The Upper bound of the number of jumps
- The Pagoda function

2.2 Upper Bound of the Number of Jumps

In this section, we will discuss this method and how we used it within a backtrack search. The main goal of this technique is, for a feasible problem, to have a list of upper bounds for each of the jumps. This will allow us to stop exploring solutions for which the upper bound of a jump has already been exceeded. One way to achieve this goal is to use Integer Programming. Indeed, for each jump j, we can try to maximize the number of moves using jump j under same constraints as our initial integer program, which means:

Upper Bound determination for jump j - Integer program :

$$\max \ \boldsymbol{x}_j^1 + \boldsymbol{x}_j^2 + \ldots + \boldsymbol{x}_j^l \tag{2.1}$$
$$\text{s.t. } A(\boldsymbol{x}^1 + \boldsymbol{x}^2 + \ldots + \boldsymbol{x}^l) = \boldsymbol{p_s} - \boldsymbol{p_f} \tag{2.2}$$
$$\forall k \in \{1, 2, \ldots, l\} \ \ \mathbf{0} \leq \boldsymbol{p_s} - A(\boldsymbol{x}^1 + \boldsymbol{x}^2 + \ldots + \boldsymbol{x}^k) \leq \mathbf{1} \tag{2.3}$$
$$\forall k \in \{1, 2, \ldots, l\} \ \ x_1^k + x_2^k + \ldots + x_m^k = 1 \tag{2.4}$$
$$\forall k \in \{1, 2, \ldots, l\} \ \ \boldsymbol{x}^k \in \{0, 1\}^m \tag{2.5}$$

This formulation is quite interesting for two reasons:

- First, as the constraints are the same as our initial integer program, it is obvious that our peg solitaire problem is feasible if and only if the Upper Bound Program is feasible for each jump j.
- Second, once this program is solved for each jump j, we are sure that there is no feasible sequence that contains more than the optimal number found for each j, which enables us to develop our pruning technique.

However, as we still have the same variables and constraints, this Upper Bound program has the same size as our initial integer program, which makes the resolution very long. That is why we had to relax this problem to make it more easily solvable. To do so, we start by replacing $x_j^1 + x_j^2 + \ldots + x_j^l$ by x_j for each $j \in \{1, 2, ..., m\}$. Plus, we can drop constraints (9), (10) and (11) seen above. The resulting integer program is the following:

Relaxed Upper Bound determination for jump j - Integer program :

$$\max \ \boldsymbol{x}_j \tag{2.6}$$
$$\text{s.t. } A\boldsymbol{x} = \boldsymbol{p_s} - \boldsymbol{p_f} \tag{2.7}$$
$$\forall j \in \{1, 2, \ldots, m\} \ \ \boldsymbol{x}_j \in \mathbb{N}^* \tag{2.8}$$

We end up having a program with 76 integer variables and 33 constraints, which helps us a lot in our efficiency quest. However, as we dropped many constraints, the optimal value found by this program is only an upper bound of the optimal value we are looking for for each jump. Furthermore, feasibility of this program does not guarantee the feasibility of our peg solitaire problem.

Then, we implemented a Python function in order to use this method. For that, we created a `solveRUB()` function which takes as input:

- **jump**: (integer) the jump on which we use the RUB integer program
- **starting_config**: (list) the starting configuration of our peg solitaire problem
- **finishing_config**: (list) the finishing configuration of our peg solitaire problem

This function returns the optimal value found if the RUBj problem is feasible, and `None` otherwise. We described in figure 2.2 the result of `solveRUB()` function with our given initial and final configurations.

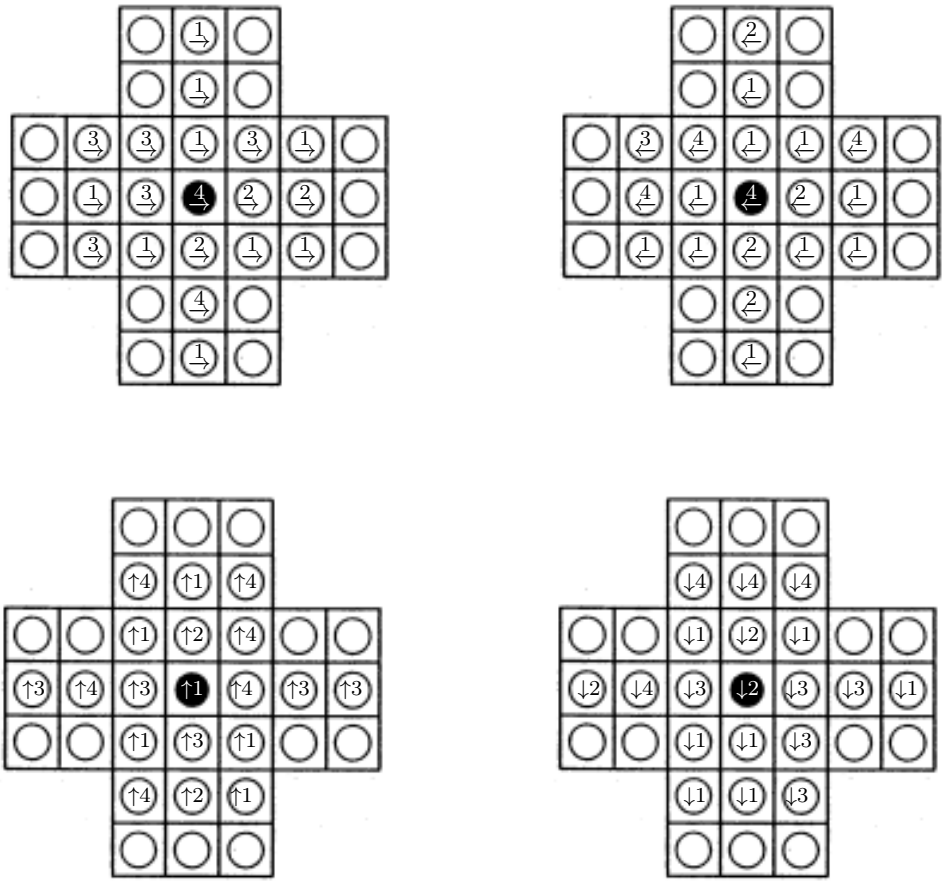


Figure 2.2: Upper bounds given by our algorithm

Moreover, we developed a second pruning technique based on *Pagoda* functions.

2.3 Pagoda function

Basically, a *Pagoda function* is a real valued function defined on the board holes that *cannot increase* during the game. Mathematically, a *pag* function $\{1, 2, ..., n\} \rightarrow \mathbb{R}$ verifies that for each three consecutive holes $\{i_1, i_2, i_3\}$:

$$pag(i_1) + pag(i_2) \geq pag(i_3)$$
$$pag(i_3) + pag(i_2) \geq pag(i_1)$$

Therefore, for a given configuration $\boldsymbol{p} \in \{O, 1\}^n$, we consider that:

$$pag(\boldsymbol{p}) = \sum_{i=1}^{n} pag(i) \times p_i$$

Actually, if we get a configuration $\boldsymbol{p'}$ by applying a jump to a configuration $\boldsymbol{p}$, we have that: $pag(\boldsymbol{p}) \geq pag(\boldsymbol{p'})$. Let's explain why with the following example, in which we assign only three holes with a pagoda function values.

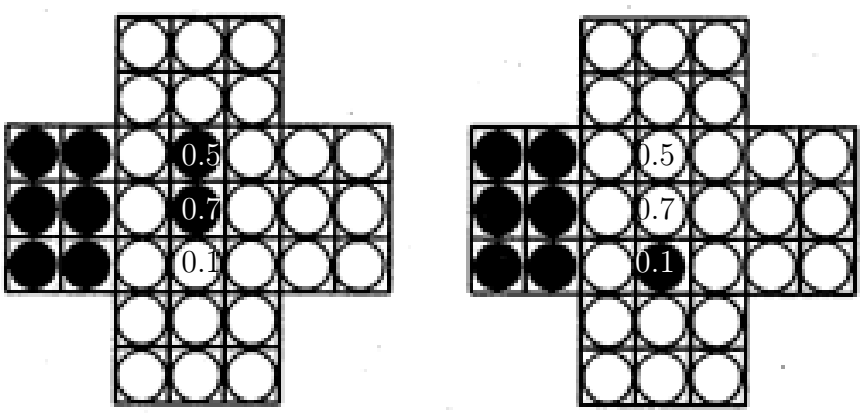


Figure 2.3: Pagoda function illustration

For the left configuration, the pagoda value is: $pag(\boldsymbol{p_s}) = 0.5 + 0.7 = 1.2$. Once we make a jump, the pagoda value becomes: $pag(\boldsymbol{p_f}) = 0.1 \leq 0.5 + 0.7$. Hence, it is obvious that every jump reduces the pagoda function value. Therefore, we can derive that for a given start and end configuration, respectively $\boldsymbol{p_s}$ and $\boldsymbol{p_f}$, we have:

$$pag(\boldsymbol{p_f}) \geq pag(\boldsymbol{p_s})$$

Thus, no given intermediate configuration can have a pagoda value strictly greater than the final configuration's pagoda value. Otherwise, we stop extending the partial solution looking for the final configuration. Furthermore, there are many different pagoda functions and each one can be more or less efficient regarding the configuration that we consider. For that reason, we decided to use three differently sparse pagoda functions containing only zeros and ones (Figure 2.4), as it is advised in [1].

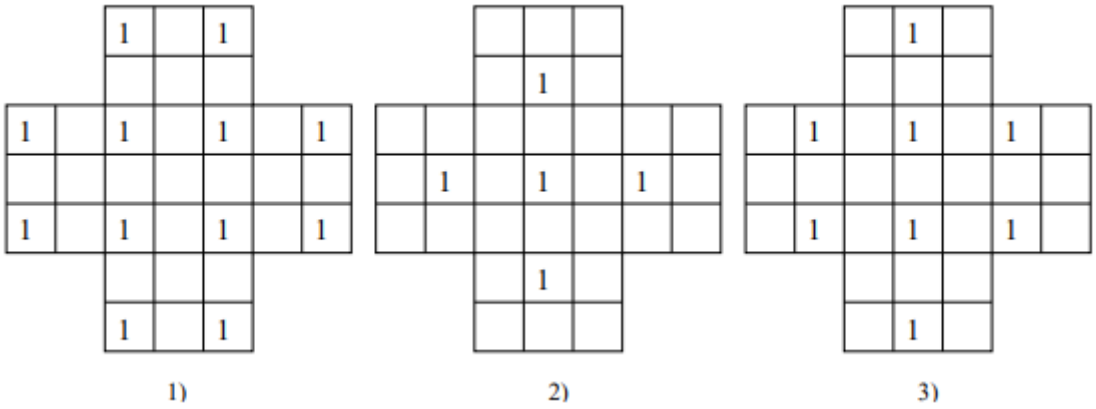


Figure 2.4: Three Pagoda functions used

2.4 Algorithmic implementation

Now that we have defined these two pruning techniques, we can start implementing our backtrack algorithm. We defined for that a `backtrack_peg_solver()` function which takes as input:

- **starting_config**: (list) the starting configuration of our peg solitaire problem
- **finishing_config**: (list) the finishing configuration of our peg solitaire problem

This algorithm returns the path as a list of successive jumps if the problem is feasible, and `None` otherwise. To do that, the steps that we follow are:

1. Initializing a **hash table**: this feature is very important in making our algorithm more efficient, as it allows us to store all the configurations that we have already been through. Hence, if we bump into an already stored configuration, there is no need to continue exploring it.
2. Computing all **upper bounds**: we use the `solveRUB()` defined earlier to create a list containing the upper bound of all the 76 jumps.
3. Using a `search()` function: this function returns 0 if the problem is unfeasible, and 1 otherwise. Let's delve now into this function's details.

`The search()` function is based on a recursive algorithm which takes as input:

- **start**: an intermediate configuration from which we try to reach the final configuration
- **end**: the final configuration that we are looking for. Hence, this input remains the same for all the `search()` function calls.
- **upper_bound**: a list of upper bounds being updated at each step, as the upper bound depend on the start and end configuration considered.
- **path**: a list containing the jumps of the solution being explored.

- **dictionary**: a hash table with all solutions that has been already computed.

The algorithm process is the following:

1. Check if we have reached the recursion termination: which means checking if the start and end configurations are equal.

2. If the start configuration still has more pegs, we use the **pagoda function technique**: if one of the three pagoda functions implemented verifies that the start pagoda value is less than the end pagoda value, we return `False` and the problem has no feasible solution.

3. Once the pagoda constraints are verified, we start our depth-first exploration of possible solutions. Indeed, we consider a given jump and his upper bound. If the upper bound is strictly positive, we make this jump and we update the start configuration. Then, if the new start configuration is already in our hash table, we do not need to call search() for this configuration again. If it is not the case, we recursively call `search()` with the updated *start* configuration.

2.5 Computational results

While evaluating the speed of our program, we used a personal computer with an `Intel Core i5` 2.50 GHz CPU and 4Go memory. For this specific problem which we are studying, the program returned that the problem is feasible along with the ordered jumps that allow to reach the final configuration from the starting one. However, it took about 30 minutes to return the result, which is quite long.

Another key issue is the order of jumps being considered. Indeed, for a given problem, there must be a most efficient way to explore jumps during the backtrack search. For this reason, we decided to shuffle columns of A matrix, in order to randomly explore jumps among the 76 possible ones while seeking for the most efficient sequence of jumps. On Jupyter, we used the function `numpy.random.shuffle()` to shuffle the columns of A.

However, we wanted to record the most efficient columns orders and have a repeatable result. For this reason, we used the function `numpy.random.seed()`. For instance, by running `numpy.random.seed(seed = 1)`, the result of `numpy.random.shuffle()` on the columns of A will always produce the same columns order. Hence, our idea was to test various seed values, and seek the fastest one. We did that by implementing a function `seed_exploration()` which tested our algorithm for various seed values and stopped it if it took too much time (more than 30 seconds). Then, we figured out that for `seed = 82`, we had a quite fast backtrack search as it took **less than 4 seconds**. The result is visible in figure 2.6.

```
np.random.seed(seed=13)
s = np.arange(A.shape[1])
np.random.shuffle(s)
A = A[:,s]
```

Figure 2.5: Shuffling A columns

```
Feasible problem
The list of moves composing the feasible solution is :
--> [61, 47, 13, 54, 32, 55, 12, 32, 63, 12, 21, 34, 32, 56, 72, 62, 45, 2, 22, 57, 41, 19, 57, 25, 50, 3, 5, 25, 67, 53, 29]
The elapsed computing time is : 3.25 s
```

Figure 2.6: Result and speed of our algorithm for `seed = 82`

```
Feasible problem
The list of moves composing the feasible solution is :
-----> [29, 9, 51, 33, 56, 55, 70, 72, 61, 47, 62, 4, 43, 35, 69, 59, 0, 2, 44, 54, 13, 15, 35, 53, 43, 59, 24, 22, 2, 60, 8]
The elapsed computing time is : 18.71 s
```

Figure 2.7: Result and speed of our algorithm for `seed = 13`

Finally, from the solution list obtained, we decided to give a graphical representation of the moves sequence. For this reason, we created a function `print_solution()` which allows to print all the moves in the order from the start configuration until the finishing one. You can find two examples in appendix A.

Conclusion

While solving the English Peg Solitaire problem, we have seen that the Integer Programming approach was pretty inefficient. Indeed, its spatial and time complexity are too high to be solved by any usual personal computer.

The main challenge was then to relax the IP formulation and to focus on specific areas of the search space to avoid computing useless paths, by implementing a backtrack search algorithm. Then, we experimented two different heuristics, which are the upper bound of jumps and the Pagoda function approach. Furthermore, using a hash table to store intermediate results was key to improve the algorithm's performance, as we ended up solving the problem in very reasonable time, which is less than 4 seconds.

Finally, we think that methods and lessons learned from this project are very interesting and innovative and they can be generalized to any other "planning" problem.

Appendix A - Program results

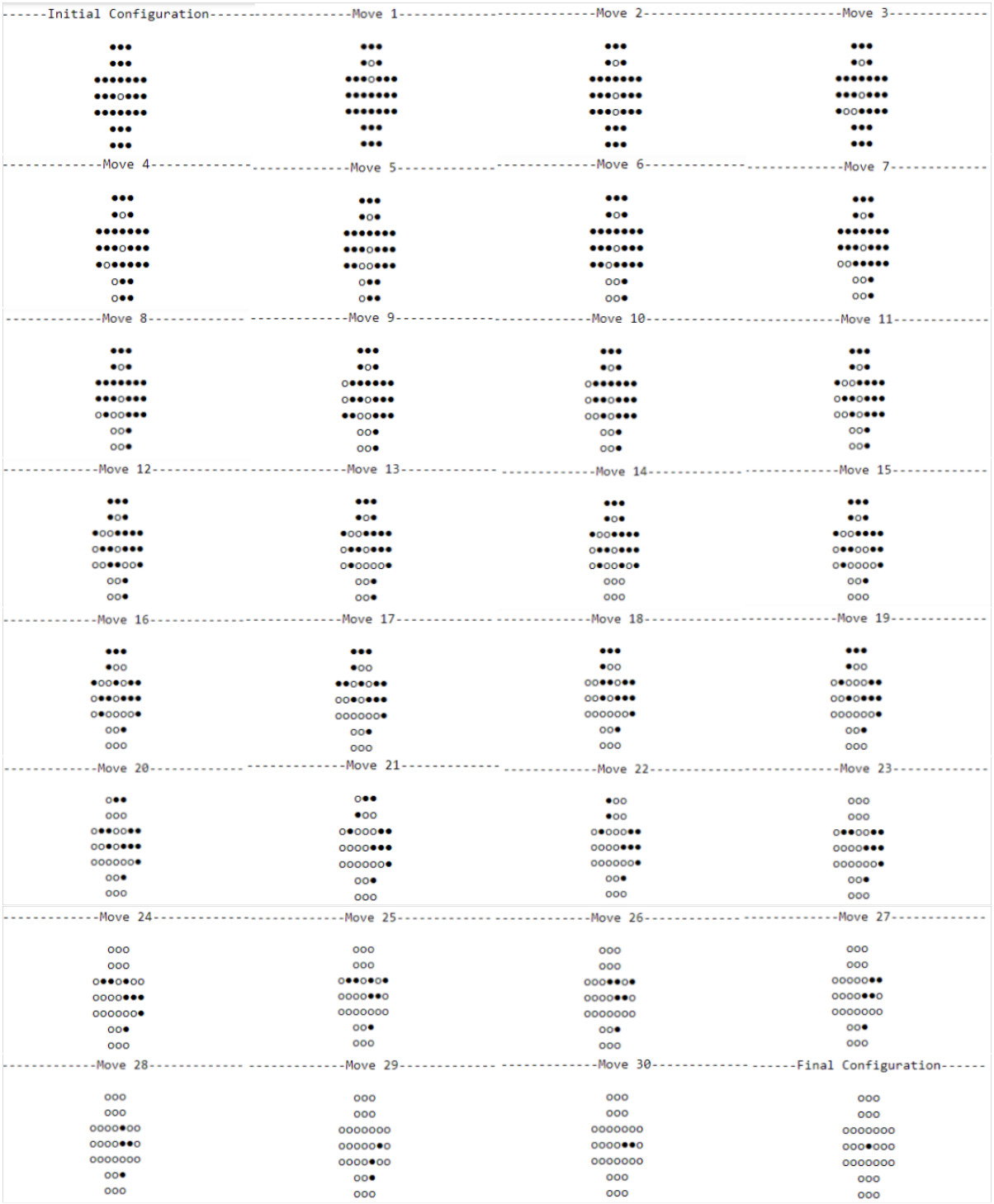


Figure A.1: Sequence of jumps for `seed = 82`

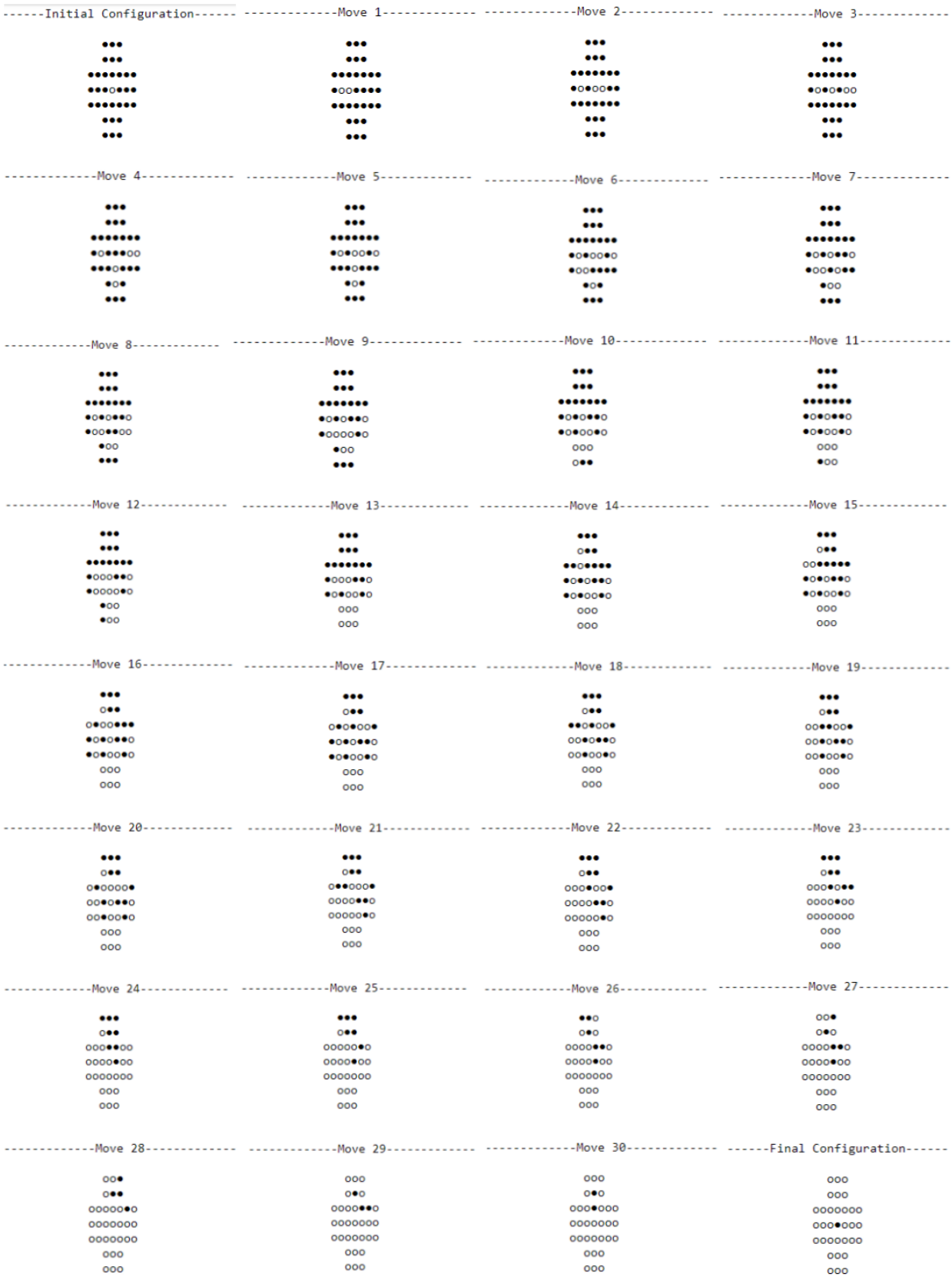


Figure A.2: Another Sequence of jumps for `seed = 13`

Bibliography

[1] Conway Berlekamp, J Conway, and RK Guy. *Guy, Winning Ways for your Mathematical Plays, Vol. 2*. 1982.

[2] Christopher Jefferson et al. "Modelling and solving English peg solitaire". In: *Computers & Operations Research* 33.10 (2006), pp. 2935–2959.

[3] Masashi Kiyomi and Tomomi Matsui. *Integer programming based algorithms for peg solitaire problems*. Springer, 2000.