Intelligent Strategies for Several Zero-, One- and Two-Player Games

Mugurel Ionut Andreica, Nicolae Tapus

Intelligent Strategies for Several Zero-, One- and Two-Player Games

Mugurel Ionut Andreica, Nicolae Tapus
Politehnica University of Bucharest, Computer Science Department, Bucharest, Romania
{mugurel.andreica, nicolae.tapus}@cs.pub.ro

Abstract

In this paper we present efficient and intelligent strategies for several zero-, one- and two-player games. Most of the games have been studied before or are related to other well-known games, but we present improved algorithmic techniques for playing them optimally. The main techniques we employed are dynamic programming, the Sprague-Grundy game theory and pattern extraction. We also make use of elements from computational geometry, like orthogonal range searching data structures.

1. Introduction

In computer science, games have constituted a major motivation for developing intelligent systems and efficient algorithmic techniques. The uprising game theory provides the means for analyzing complex interactions between rational (and/or economic) agents and for implementing strategies which maximize their revenues. In this paper we consider impartial games where full information is available, for zero-, one- and two-player games. The zero-player games do not involve the decisions of a player and are used for modeling the evolution of natural states. One-player games usually ask the player to optimize the usage of some resource, based on several constraints regarding the actions which can be performed. The two-player games we consider can be solved by traditional means, but also exhibit some unexpected patterns, which are helpful in devising more efficient game strategies. This paper is organized as follows. In Section 2 we discuss zero-player games. In Sections 3 and 4 we present several one- and two-player games. In Section 5 we present related work and we conclude.

2. Zero-Player Games

In this section we consider a particular one-dimensional cellular automaton (which evolves without any player's intervention) for which we provide a method which efficiently evaluates its state after any given number m of time steps. The cellular automaton consists of n cells (numbered from 0 to n-1, from left to right) and, at any time moment t, the state of each cell i (q(i,t)) can be 0 or 1. At each time step, every pair of adjacent cells $i \geq 0$ and $i+1<n$, such that $q(i,t)=0$ and $q(i+1,t)=1$ exchange their states (the 0 and 1 are swapped). The final state of such an automaton is reached after $T=O(n)$ steps, when all the 0s are to the left of all the 1s. A naive algorithm for computing the state of the automaton after every number $m \leq T$ of steps would take $O(n \cdot m)$ time. We will now provide an O(n) algorithm for this problem. We will assign a number from 0 to nz-1 to each zero state of the automaton, in a left to right order (nz is the total number of zero states). The i^{th} zero is located at the cell c(i). It is obvious that all the zeroes "move" to the left and that, in the final (stable) state, the i^{th} zero will be located at cell i. It is also obvious that the i^{th} zero ($i \geq 1$) will not reach cell i before the $(i-1)^{th}$ zero reaches cell i-1. During every time step, a zero state performs an action: it either "moves" one cell to the left (if the state of the cell to the left is 1) or "waits" (if the state of the cell to the left is 0). For each zero state i, we will determine the sequence of $na(i) \geq 0$ actions $a_{i,1}$, $a_{i,2}$, …, $a_{i,na(i)}$ performed until it reaches its final cell. The sequence will be represented in reverse order, i.e. $a_{i,na(i)}$ is the action performed during the first time step and $a_{i,1}$ is the last action performed. Based on this sequence of actions, we will be able to determine in O(1) time the cell where each zero is located after m time steps. For the zero state numbered with 0, its sequence of actions consists of $na(0)=c(0)-0$ "moves": a_j="move" ($0 \leq j \leq na(0)$). We will determine the sequence of actions for each zero state, in increasing order of their assigned number. If $c(i)=c(i-1)+1$, then the sequence of actions for the i^{th} zero state is identical to the one for the $(i-1)^{th}$ zero state, except that the first action performed is a "wait". Thus, we have: $na(i)=na(i-1)+1$, $a_{i,j}=a_{i-1,j}$ ($1 \leq j \leq na(i)-1$) and $a_{i,na(i)}$="wait". If $c(i)>c(i-1)+1$, then the first $d=c(i)-c(i-1)-1$ actions of the i^{th} zero state will

be "moves". We need to find out if the i^{th} zero "catches up" with the $(i-1)^{th}$ zero before the $(i-1)^{th}$ zero reaches its final cell and if it does, after how many time steps this situation occurs. If the $(i-1)^{th}$ cell performs less than d "waits", then the i^{th} zero does not catch up with the $(i-1)^{th}$ zero and the actions performed by it will be: $a_{i,1}=a_{i,2}=\ldots=a_{i,c(i)-i}=$ "move". If the i^{th} zero "catches up" with the $(i-1)^{th}$ zero after t time steps (i.e. after t time steps, it is located immediately to the right of the $(i-1)^{th}$ zero), then we have $na(i)=na(i-1)+1$, $a_{i,na(i)}$="move", …, $a_{i,na(i)-(t-1)}$="move", $a_{i,na(i)-t}$="wait" and $a_{i,na(i)-j}=a_{i-1,na(i)-j}$, for $t+1\leq j\leq na(i)-1$. In order to determine the value of t efficiently, we will also compute two arrays for each zero state: *totalWaits[i,j]*=the number of "wait" actions in the set $\{a_{i,1}, a_{i,2}, \ldots, a_{i,j}\}$ and *nextWait[i,j]*=the first "wait" action $a_{i,j'}$, $j'\leq j$. The algorithm maintains a stack *a* with the sequence of actions corresponding to the $(i-1)^{th}$ zero state and transforms this stack into the sequence of actions of the i^{th} zero state. Similarly, the arrays *totalWaits* and *nextWait* will also only be transformed from the $(i-1)^{th}$ zero to the i^{th} zero.

3. One-Player Games

3.1. 1D Push-*

Push-* [2] is a simplified version of the well-known 2D game Sokoban. In this section we consider the one-dimensional version of Push-*, with several additions. There are N squares on a linear board, numbered from 1 to N (from left to right). Some of the squares contain blocks, while others are empty. A robot starts in square 1 and must arrive to square N with minimum consumption of energy. In order to achieve this, the robot can make the following moves: walk, jump and push. A walk consists of moving from the current square to the left or to the right if the destination square is empty. If the robot's square is i and square i+1 contains a block, the robot may push that block one square to the right (together with all the blocks located between positions i+2 and the first empty square to the right of i+1); obviously, an empty square must exist somewhere to the right of position i+1. After the push, the robot's position becomes i+1. In a similar manner, the robot can push blocks to the left. The robot can also jump any number Q ($1\leq Q\leq K$) of squares to the right (left) if the previous $(K-1)\geq 1$ moves consisted of walking to the right (left). Each type of move consumes a certain amount of energy. We will find the minimum energy strategy with a dynamic programming algorithm. We compute a table *E[i,j]*=the minimum energy consumed in order to have the robot located at square i and having j empty squares to the left (i.e., the squares i-1, i-2, …, i-j are empty). Furthermore, the robot has not yet reached any square k>i (thus, all these squares are in the same state as in the beginning). In order to justify this approach, we will consider the squares grouped into intervals of consecutive empty squares. Let's number these intervals with consecutive numbers, from left to right. If the robot reaches a square inside an interval *X*, then an optimal strategy will never contain moves which bring the robot to an interval *Y*<*X*. Thus, when the robot arrives in a square i inside an interval *X*, all the squares k>i are in the initial state. This way, we can consider only sequences of moves which are local to the interval of consecutive empty squares into which the robot resides. The outcome of these moves should be that the player reaches another interval *Y*>*X* (or another square k>i). For each state (i,j), we need to consider only $O(N^2)$ sequences of moves, which will improve the value of some states (i',j'), i'>i. These sequences consist of travels (walks+jumps), walks and pushes (to both sides of the interval of empty squares). Since there are $O(N^2)$ possible states, the time complexity will be $O(N^4)$.

3.2. Candy Collector

We have a complete directed graph with N vertices, numbered from 1 to N. The player is initially located at vertex 1. For each ordered pair of vertices (i,j), the time required to travel from i to j, $tr_{i,j}$, is given. At certain time moments, boxes of candies may appear in the vertices of the graph. There are M boxes overall and for each box of candy k, the time moment when it appears, ta_k, the vertex where it appears, v_k, and the number of candies in the box, c_k, are known. All the time moments are considered to be integers. At each moment t, the player may either stay in its current position (vertex) i or may start traveling towards another vertex j (which he/she reaches at time moment $t+tr_{i,j}$). The candies inside a box k can be collected by the player only if the player is located at vertex v_k at the moment the candy box appears (ta_k) or if the player just arrives at the vertex at that moment. The purpose of the game is to collect as many candies as possible. An optimal strategy can be found by using dynamic programming. We sort the candy boxes in increasing order of their moment of appearance. Thus, box k appears after (or at exactly the same time as) any box p<k. For each candy box k, we compute $C_{max}[k]$=the maximum number of candies which the player can collect if at time ta_k he/she arrives (or is located) at vertex v_k (and, thus, collects the candies in box k). We also consider a virtual box k=0 with $c_0=0$ candies, appearing at $v_0=1$ at $ta_0=0$. We have $C_{max}[0]=0$ and

$$C_{max}[k] = c_k + \max\left\{\begin{matrix} Cmax[p], \text{if } p < k \text{ and } tr_{v_p,v_k} \le ta_k - ta_p \\ -\infty \end{matrix}\right\} \quad (1)$$

The maximum number of candies which can be gathered is the maximum value in the array C_{max}. The time complexity of this algorithm is $O(M^2)$. We will now consider the case when M is large: for instance, M>N and/or $M>T_{max}$, where T_{max} is an upper limit for the maximum travel time between any two vertices. We will compute the same values as above, but we will make the following observation: if $v_p=v_k$ (p<k), then $C_{max}[p] \le C_{max}[k]$. For each vertex i, we will maintain a list with all the candy box numbers which appeared at vertex i, sorted in chronological order. Let this list be cb(i,1), cb(i,2), …, cb(i,ncb(i)), where ncb(i) is the number of candy boxes which appeared at vertex i (so far). When computing $C_{max}[k]$ for a candy box k, we will iterate over all the vertices of the graph. For each vertex i, we will find the last candy box cb(i,j), such that $tr_{i,vk} \le ta_k - ta_{cb(i,j)}$ and set $C_{max}[k]=\max\{C_{max}[k], c_k+C_{max}[cb(i,j)]\}$. Since the candy boxes cb(i,1), …, cb(i,ncb(i)) are sorted such that $ta_{cb(i,1)} < \ldots < ta_{cb(i,ncb(i))}$, we can perform a binary search in order to find the candy box cb(i,j). The time complexity becomes $O(M \cdot N \cdot \log(M))$. After computing $C_{max}[k]$, we add k at the end of the candy box list of the vertex v_k. When the maximum travel time between any two vertices i and j ($tr_{i,j}$) is less than (or equal to) a small value T_{max}, we can improve the algorithm further. For each vertex i, we will maintain a value $T_{last}[i]$=the last time moment when a candy box appeared at vertex i. We will also maintain a table MaxC[i,t], with $0 \le t \le T_{max}$, representing the maximum number of candies the player can gather if at time $T_{last}[i]$-t he/she is located at vertex i. Initially, $T_{last}[i]=0$, for all the vertices i, and $MaxC[i,t]=-\infty$, except for MaxC[1,0], which is 0. With these values, we can compute $C_{max}[k]$ in $O(M \cdot (N+T_{max}))$. If the graph's vertices are points on the OX axis (each point i having a coordinate x_i) and the travel time between two vertices i and j is the difference between their coordinates ($tr_{i,j}=|x_i-x_j|$), we can consider that the *OY* axis corresponds to time. With this representation, each candy box k is a point with coordinates (x_{vk}, ta_k). When computing the value $C_{max}[k]$ of the candy box k, we are interested in the C_{max} values of candy boxes p<k whose coordinates have the following property: $|x_{vp}-x_{vk}| \le ta_k - ta_p$. This equation defines a rectangular quarter-plane, with the origin in (x_{vk}, ta_k). By rotating all the points associated by 45 degrees around the origin, each candy box is assigned some new coordinates (x_k', y_k'). With the new coordinates, the condition for a candy box p<k to be considered when computing $C_{max}[k]$ is: $x_p' \le x_k'$ and $y_p' \le y_k'$. The quarter-plane is now aligned with the *OX'* and *OY'* axes. If we consider the value $C_{max}[k]$ of a candy box k to be the weight of the point (x_k', y_k'), we are interested in finding the maximum weight of a point located inside a quarter-plane. We can use orthogonal range search data structures, like 2D range trees, for which range queries and updates take $O(\log^2 M)$ each.

4. Two-Player Games

4.1. K in a Row

There are N empty (unoccupied) squares on a linear board, numbered from 1 to N (from left to right). At each turn, a player must occupy K≥1 consecutive unoccupied squares. The first player which cannot perform a move when its turn comes loses the game. The game can be analyzed using the Sprague-Grundy game theory. For every number i (0≤i≤N), we will compute G(i)=the Grundy number of a game state consisting of i consecutive empty squares. We have G(i)=0, for i<K. For i≥K, we will generate the set of all the successor states. There are i-K+1 possible moves, according to the first occupied square. If the first occupied square is j (1≤j≤i-K+1), then, after occupying the squares j, j+1, …, j+K-1, we have j-1 empty squares to the left and i-j-K+1 empty squares to the right. Thus, we now have a sum of two independent games, composed of j-1 and i-j-K+1 squares. The Grundy number of the sum of the two games is G(j-1) xor G(i-j-K+1). The set of Grundy numbers of all the successor states is SG={G(0) xor G(i-K), G(1) xor G(i-K-1), …, G(i-K) xor G(0)} and G(i)=mex(SG) (mex = minimum excluded value). Although the game can be analyzed this way, the complexity of computing all the Grundy numbers is $O(N^2)$ overall. Let's consider now the case K=1. It is obvious that the first player (the one performing the first move) will win only if N is an odd number; otherwise, the second player will win. For K≥2, we will focus on the losing states (with Grundy numbers equal to 0). We will consider the sequence of losing states s_0, s_1, s_2, …, starting from s_0=K-1 (the states with less than K-1 squares are also losing states, but they can be trivially handled). Starting from this sequence, we consider the sequence of differences between two consecutive losing states, i.e. a sequence d_1, d_2, …, where $d_i=s_i-s_{i-1}$. For K=2, the sequence of differences has a prefix of length 8 (4, 4, 6, 6, 4, 4, 6, 4) and a period of length 5 afterwards (4, 12, 4, 4, 10). Using this information, we can find out in O(1) time whether a given state is a winning or losing one. For K≥4, we noticed another interesting pattern regarding the sequence of differences: the first 12 differences are always $d_1=2 \cdot K$, $d_2=2 \cdot K$, $d_3=4 \cdot K-2$, $d_4=4 \cdot K-2$, $d_5=4 \cdot K$, $d_6=4 \cdot K-2$, $d_7=8 \cdot K-2$, $d_8=4 \cdot K-2$, $d_9=8 \cdot K$, $d_{10}=8 \cdot K-2$,

$d_{11}=16\cdot K-6$, $d_{12}=4\cdot K$. We could not find any other pattern after d_{12}. However, with this pattern, we can analyze in O(1) time any state between 1 and $69\cdot K-19$.

4.2. Collect an Even/Odd Number of Objects

There is one pile containing N objects (N is odd). Two players perform moves alternately. When its turn comes, a player may remove from the pile any number of objects x between 1 and K (if there are at least x objects in the pile). The player keeps the objects he/she removed and adds them to the objects removed during previous moves. When the pile becomes empty, each player counts the number of objects he/she gathered from the pile during the game. The winner of the game is the player who gathered an even number of objects. Here we can use dynamic programming, by computing two sets of values: *win[0,i]* and *win[1,i]*. *win[0,i]* is 1, if the pile contains i objects, the winner must gather an even number of objects and the player whose turn is next has a winning strategy (and 0, otherwise); *win[1,i]* is defined similarly, except that the winner must gather an odd number of objects. We have *win[0,0]=1* and *win[1,0]=0*. For $1\le i\le N$, we have:

$$\text{win}[0,i]=\begin{cases}1, \text{if } \exists(1\le c\le \min\{i,K\}) \text{ such that}\\ \text{win}[((c \bmod 2)+1+((i-c) \bmod 2)) \bmod 2, i-c]=0\\ 0, \text{otherwise}\end{cases} \quad (2)$$

$$\text{win}[1,i]=\begin{cases}1, \text{if } \exists(1\le c\le \min\{i,K\}) \text{ such that}\\ \text{win}[((c \bmod 2)+((i-c) \bmod 2)) \bmod 2, i-c]=0\\ 0, \text{otherwise}\end{cases} \quad (3)$$

If win[0,N]=1, then the first player has a winning strategy; otherwise, the second player has one. An algorithm implementing the equations above has time complexity $O(N\cdot K)$ and considers the number of objects in the pile in increasing order. The time complexity can be improved to O(N), by maintaining a table *last[x,y,z]* ($0\le x,y,z\le 1$), with the following meaning: the last value of i (number of objects in the pile) such that: the parity of the number of objects gathered by the winner is x (0 for even, 1 for odd), y=((the number i of objects in the pile) mod 2) and z=win[x,i]. The new equations for win[0,i] and win[1,i] and the algorithm are given below:

$$\text{win}[0,i]=\begin{cases}1, \text{if } (i-\text{last}[((i-1) \bmod 2),(i \bmod 2),0])\le K\\ 1, \text{if } (i-\text{last}[((i-1) \bmod 2),((i-1) \bmod 2),0])\le K\\ 0, \text{otherwise}\end{cases} \quad (4)$$

$$\text{win}[1,i]=\begin{cases}1, \text{if } (i-\text{last}[(i \bmod 2),(i \bmod 2),0])\le K\\ 1, \text{if } (i-\text{last}[(i \bmod 2),((i-1) \bmod 2),0])\le K\\ 0, \text{otherwise}\end{cases} \quad (5)$$

The values win[0,i] and win[1,i] present some unexpected patterns. For even K, we have *win[0,N]=0*, only if *(N mod (K+2)=1)*. For odd K, we have *win[0,N]=0*, only if *(N mod (2·K+2)=1)*. We should notice that, by computing the win[0,i] and win[1,i] values, we also solved the version of the game in which the winner has to gather an odd number of objects. The values of win[1,N] exhibit similar patterns. For odd K, we have *win[1,N]=0*, only if *(N mod (2·K+2)=(K+2))*. For even K, *win[1,N]=0*, only if *(N mod (K+2)= (K+1))*. Similar rules can be developed for win[0,N] and win[1,N] when N is even, but in this case both players may win the game. For even N and odd K, *win[0,N]=0*, only if *(N mod (2·K+2)=(K+1))* and *win[1,N]=0*, only if *(N mod (2·K+2)=0)*. For even N and even K, win[0,N] is always 1 and *win[1,N]=0*, only if *(N mod (K+2)=0)*.

5. Related Work & Conclusions

Cellular automata were made popular by Conway's Game of Life [1], but have since emerged as an important scientific topic [3]. Single-player games are very popular; some of them which are closely related to the ones studied in this paper are presented in [2,5]. The Sprague-Grundy theory [6] for two-player impartial games is the best known for analyzing *normal play* games (those where the winner makes the last move). In [4], periodicity and arithmetic-periodicity aspects of some hexadecimal two-player impartial games are considered. In this paper we discussed several zero-, one- and two-player games, for which we identified new, unexpected patterns and developed new techniques for computing optimal strategies.

6. References

[1] M. Gardner, "Mathematical Games: The fantastic combinations of John Conway's new solitaire game 'Life'", *Scientific American 223*, 1970, pp. 120-123.

[2] E. D. Demaine, M. L. Demaine, J. O'Rourke, "PushPush and Push-1 are NP-hard in 2D", *Proc. of the 12th Canadian Conference on Computational Geometry*, 2000, pp. 211-219.

[3] E. Goles, "Parallel and Serial Dynamics in Boolean Networks", *Proc. of the 13th International Workshop on Cellular Automata*, 2007.

[4] S. Howse, R. J. Nowakowski, "Periodicity and arithmetic-periodicity in hexadecimal games", *Theoretical Computer Science*, vol. 313, 2004, pp. 463-472.

[5] E. D. Demaine, M. L. Demaine, H. A. Verrill, "Coin-Moving Puzzles", *More Games of No Chance*, Cambridge University Press, 2002, pp. 405-431.

[6] P. M. Grundy, "Mathematics and Games", *Eureka 2*, 1939, pp. 6-8.