Teaching the Backtracking Method using Intelligent Games

Anca Andrei and W. Ted Mahavier
Department of Mathematics, Lamar University,
Beaumont, Texas, 77710
aandrei@lamar.edu, ted.mahavier@lamar.edu

Abstract. Teaching programming techniques has always been a challenge. We exhibit an innovative way of teaching the backtracking programming strategy using educational games, in particular Sudoku. Compared to traditional ways of teaching backtracking, we used a new pseudo-code that corresponds to the solution tree of the educational game. Most of the current textbooks present the traditional way to teach backtracking by showing the recursive call at the end of the `backtrack()` method.

We will show an alternative to describing the `backtrack()` method by extending the solution with a choice, followed by the recursive call, which is in turn followed by an undo of that choice (that is, the backtracking step). As such, a path in the solution tree will correspond to a sequence of recursive calls of the `backtrack()` method. In this way, anyone who knows Sudoku rules or any other educational game (chess, eight queens, the knight's tour problem, etc.) will most likely understand the backtracking strategy. Another contribution of our work is a mathematically sound method to transform a random Sudoku grid into a similar one which accepts only one Sudoku solution.

Keywords: backtracking, educational games, teaching, programming, algorithm

1. Introduction

An algorithm design technique (also known as strategy or paradigm or method) is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing. There are many algorithm design techniques used to solve computer science problems, such as the exhaustive search, the decrease-and-conquer approach, divide-and-conquer approach, the transform-and-conquer, the dynamic programming, the greedy technique, the backtracking approach, and the branch-and-bound technique [McConnell; 2007], [Levitin; 2012], [Cormen, Leiserson, Rivest, Stein; 2009], [Johnsonbaugh and Schaefer; 2010], [Miller and Boxer; 2013], and [Shackelford; 1997].

The exhaustive search technique suggests generating all candidate solutions and then identifying the one(s) having the desired property. Backtracking is a more intelligent variation of this approach. The principal idea is to construct solutions one component at a time and evaluate such partially constructed candidates as follows. We illustrate backtracking by constructing a tree of choices called the *search-space tree*. Its root represents an initial state before the search for a solution begins. The nodes of the first level in the tree represent the choices made for the first component, and so on. A node in the search-space tree is said to be *promising* if it corresponds to a partially constructed solution that may still lead to a complete solution; otherwise, it is called *non-promising*. Leaves represent either non-compromising dead ends or complete solutions found by the algorithm, usually using a depth-first search. If the current node is promising, its child is generated by adding the first remaining legitimate option for the next component of a solution.

A preliminary version of this paper was awarded with the First Prize at the Student Poster Competition of the ACET'2013 conference. The Judges encouraged us to submit a journal version to ACET.

If the current node turns out to be non-promising, the algorithm backtracks to the node's parent to consider the next possible option for its last component; if there is no such an option, it backtracks one more level up the tree, and so on. Finally, if the algorithm reaches a complete solution to the problem, it either stops (if just one solution is required) or continues searching for other possible solutions.

2. Sudoku: Our motivating problem for teaching backtracking

When teaching the backtracking strategy, many problems may be used to illustrate the search-space tree and the details of the backtracking: the n-queens problem, the Hamiltonian circuit problem, the subset-sum problem, the knight's tour problem, etc. A backtracking algorithm generates a state-space tree, its nodes representing partially constructed tuples with the first *i* coordinates defined by the previous actions of the algorithm. If such a tuple (x_1, x_2, …, x_i) is not a solution, the algorithm finds the next element in S_{i+1} that is consistent with the values of (x_1, x_2, …, x_i) and the problem's constraints, and adds it to the tuple as its $(i + 1)^{st}$ coordinate. If such an element does not exist, the algorithm backtracks to consider the next value of x_i, and so on.

This presentation shows an innovative way of teaching the backtracking strategy using Sudoku. Compared to traditional ways of teaching backtracking, we used a new pseudo-code that corresponds to the solution tree of the educational game. Most of the current textbooks present the traditional way to teach backtracking by showing the recursive call at the end of the `backtrack()` method. Here is the pseudo-code of the backtracking algorithm used in most textbooks for teaching purposes. It may be called for $i = 0$, where `X[1 … 0]` represents the empty tuple.

```
Algorithm Backtrack(X[1 … i]) {
 // Input: X[1 … i] represents the first i promising components of a solution
 // Output: All the tuples representing the problem's solutions
 if (X[1 … i] is a solution) then print(X[1 … i]);
 else
     for (each element x ∈ S_{i+1} consistent with X[1 … i] and the constraints) do {
       X[i + 1] = x;
       Backtrack(X[1 … i+1]);
     }
} // end of algorithm Backtrack( )
```

Before illustrating the new pseudo-code for the Backtracking algorithm, we define the Sudoku problem and historical connotations. Sudoku was first mentioned in 1986 by the Japanese puzzle company Nikoli, under the name Sudoku, meaning digit, and became an international hit in 2005. The World Sudoku Championship and the Philadelphia Inquirer Sudoku National Championship are two annual events that give prizes to winners. The objective is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 blocks that compose the 9×9 grid contains all of the digits from 1 to 9. In fact, Sudoku is a restrictive Latin square with an additional constraint on the contents of individual blocks. For example, the same single integer may not appear twice in the same 9×9 grid row or column or in any of the nine 3×3 blocks of the 9×9 grid. Each puzzle is represented by a partially completed grid, which typically has a unique solution. However, this is not certain! The objective is to check whether a partially completed grid has indeed a unique solution.

Definition 2.1. Formally, a 9×9 grid (D stands for 'Digit') denoted by

$D_{1,1}\ D_{1,2}\ D_{1,3}\ D_{1,4}\ D_{1,5}\ D_{1,6}\ D_{1,7}\ D_{1,8}\ D_{1,9}$
$D_{2,1}\ D_{2,2}\ D_{2,3}\ D_{2,4}\ D_{2,5}\ D_{2,6}\ D_{2,7}\ D_{2,8}\ D_{2,9}$
$D_{3,1}\ D_{3,2}\ D_{3,3}\ D_{3,4}\ D_{3,5}\ D_{3,6}\ D_{3,7}\ D_{3,8}\ D_{3,9}$
$D_{4,1}\ D_{4,2}\ D_{4,3}\ D_{4,4}\ D_{4,5}\ D_{4,6}\ D_{4,7}\ D_{4,8}\ D_{4,9}$
$D_{5,1}\ D_{5,2}\ D_{5,3}\ D_{5,4}\ D_{5,5}\ D_{5,6}\ D_{5,7}\ D_{5,8}\ D_{5,9}$
$D_{6,1}\ D_{6,2}\ D_{6,3}\ D_{6,4}\ D_{6,5}\ D_{6,6}\ D_{6,7}\ D_{6,8}\ D_{6,9}$
$D_{7,1}\ D_{7,2}\ D_{7,3}\ D_{7,4}\ D_{7,5}\ D_{7,6}\ D_{7,7}\ D_{7,8}\ D_{7,9}$
$D_{8,1}\ D_{8,2}\ D_{8,3}\ D_{8,4}\ D_{8,5}\ D_{8,6}\ D_{8,7}\ D_{8,8}\ D_{8,9}$
$D_{9,1}\ D_{9,2}\ D_{9,3}\ D_{9,4}\ D_{9,5}\ D_{9,6}\ D_{9,7}\ D_{9,8}\ D_{9,9}$

is a *complete solution* if

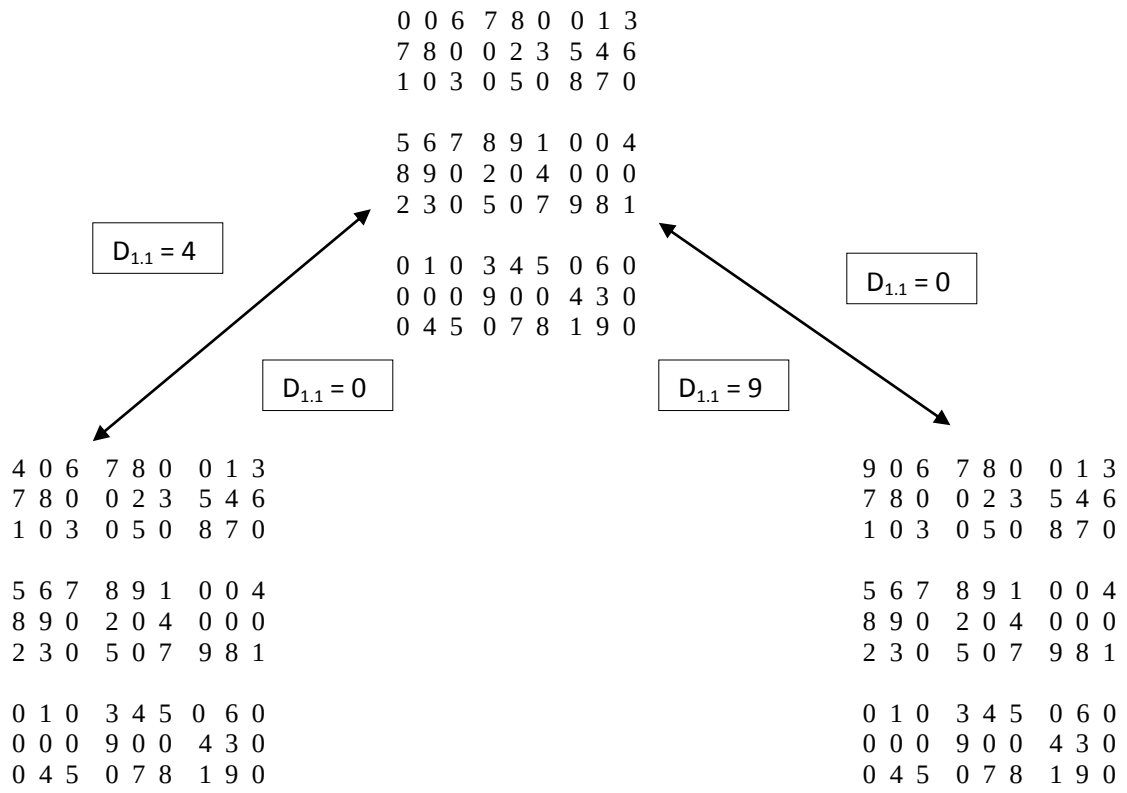
1. $1 \le D_{i,j} \le 9$ for all $1 \le i \le 9$, $1 \le j \le 9$.
2. $D_{k,i} \ne D_{k,j}$ for all $i \ne j$, $1 \le i \le 9$, $1 \le j \le 9$, $1 \le k \le 9$.
3. $D_{i,k} \ne D_{j,k}$ for all $i \ne j$, $1 \le i \le 9$, $1 \le j \le 9$, $1 \le k \le 9$.
4. Each set of digits $\{D_{i,j}, D_{i,j+1}, D_{i,j+2}, D_{i+1,j}, D_{i+1,j+1}, D_{i+1,j+2}, D_{i+2,j}, D_{i+2,j+1}, D_{i+2,j+2}\}$ contains distinct digits for all $i \in \{1, 4, 7\}$, $j \in \{1, 4, 7\}$. ■

The **(OneSudoku)** problem is then:
The input: a 9×9 partially completed grid;
The output: a completed Sudoku grid, if one exists, or the message 'no solution' if none exists.

The **(AllSudoku)** problem is then:
The input: a 9×9 partially completed grid;
The output: all completed Sudoku grids, if at least one exists, or the message 'no solution' if none exists.

Algorithm `Backtrack(X[1 … i]) {`
 // Input: `X[1 … i]` represents the first `i` promising components of a solution
 // Output: All the tuples representing the problem's solutions
 `if (X[1 … i]` is a solution`) then` print`(X[1 … i]);`
 `else`
 `for (`each element $x \in S_{i+1}$ consistent with `X[1 … i]` and the constraints`) do {`
 `X[i + 1] = x;`
 `Backtrack(X[1 … i+1]);`
 `X[i + 1] = 0;`
 `}`
`}` // end of algorithm `Backtrack( )`

The difference between our pseudo-code and the one from the [Levitin; 2012]'s version is the statement `X[i + 1] = 0;` after the recursive call. We implemented our pseudo-code in the Java programming language by constructing the state-space tree in order to lead to one solution. Here is an example of Sudoku grid and the first two levels of its state-space tree:



To the best of our knowledge, all textbooks teaching backtracking show only one edge or arc between the first node and each of the candidate solutions [McConnell; 2007] and [Levitin; 2012]. In the previous figure, these would be $D_{1,1} = 4$ and $D_{1,1} = 9$. However, no textbooks show the upward arc labeled by $D_{1,1} = 0$. We consider that showing the assignment statement $D_{1,1} = 0$ in the state-space tree is essential in building a correct tree and a sound and complete pseudo-code for the backtracking strategy. Hence, here is our version of such a sound and complete pseudo-code (we follow the same notations as [Levitin; 2012]. It can be called for $i = 0$; X[1 … 0] represents the empty tuple.

3. Towards the Uniqueness of the Sudoku Table

This section presents some results related to the uniqueness of the Sudoku grid.

Definition 3.1. Given two 9×9 Sudoku grids $A = (a_{i,j})$ and $B = (b_{i,j})$, we say that $B \Rightarrow A$ if for all i, j $\in \{1, \ldots, 9\}$, $b_{i,j} = a_{i,j}$ or $b_{i,j} = 0$. ■

If $b_{i,j}$ = '0' in a 9×9 Sudoku grid then the value of $b_{i,j}$ may be later substituted by any value from {1, 2, …, 9}. Thus, Definition 3.1 says that $B \Rightarrow A$ if the corresponding values of grid B are either equal or more general (that means '0') with the values of grid A.

Definition 3.2. Given $A = (a_{i,j})$ a 9×9 Sudoku grid, we say that A is a *complete* grid if $a_{i,j} \neq 0$, $\forall$ i $\in$ {1, 2, …, 9} and $\forall$ j $\in$ {1, 2, …, 9}. ■

Lemma 3.1. Let $A = (a_{i,j})$ and $B = (b_{i,j})$ be two 9×9 Sudoku grids such that B is a complete grid and $A \Rightarrow B$. If A is the input for the Sudoku backtracking algorithm generating all solutions, then B is one of the solutions that will provided by the Sudoku backtracking algorithm. ■

Definition 3.3. Let $A = (a_{i,j})$, $B = (b_{i,j})$, and $R = (r_{i,j})$ be three 9×9 Sudoku grids such that $R \Rightarrow A$ and

$R \Rightarrow B$. We define $R (A \square B$ to be the 9×9 Sudoku grid such that $(R (A \square B)_{i,j} = R_{i,j}$ if $a_{i,j} = b_{i,j}$ and $(R (A \square B)_{i,j} = a_{i,j}$ if $a_{i,j} \neq b_{i,j}$. ■

So far, there is no guarantee that the Sudoku backtracking algorithm will generate exactly one solution grid. We provide Theorem 3.1, which narrows the solution set down to one solution.

Theorem 3.1. Let us consider the backtracking algorithm that has as input a Sudoku grid called R. Let A and B be two arbitrary/random complete 9×9 Sudoku grids as among all solutions provided as output. If we run again the backtracking algorithm that has $(R (A \square B)_{i,j}$ as input, then:

1. A will be again a complete 9×9 Sudoku grid solution among all solutions provided as output by the backtracking algorithm,
2. B will not be a complete 9×9 Sudoku grid solution provided as output by the backtracking algorithm. ■

Here is the algorithm that finds the grid with a unique solution.

The (GridWithUniqueSolution) Algorithm

The input: The initial Sudoku grid G that has $G_1, G_2, \ldots, G_n$ the set of all Sudoku solutions;

The output: A Sudoku grid G' that will have only one solution G_1, but not the other grids $G_2, \ldots, G_n$.

The Method:

G' = G;

```
for (int i = 2; i <= n; i++)
```

G' = G' ($G_1 \square G_i$;

```
return G';
```

Theorem 3.2. Let G be the initial Sudoku grid that has $G_1, G_2, \ldots, G_n$ the set of all grid solutions provided by the **(GridWithUniqueSolution)** algorithm. Then the output of the **(GridWithUnique Solution)** algorithm is correct. Moreover, the **(GridWithUniqueSolution)** algorithm has a $O(n)$ time complexity, where n is the number of all solutions given as input of the algorithm. ■

4. Conclusion

This presentation shows an innovative way of teaching the backtracking strategy using educational games, in particular Sudoku. Compared to traditional methods for teaching backtracking, we used a new pseudo-code that corresponds to the solution tree of the educational game. In addition, we presented a mathematically sound method to transform a random Sudoku grid into a similar one which accepts only one Sudoku solution.

5. List of references

[Cormen, Leiserson, Rivest, Stein; 2009] Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein. *Introduction to Algorithms*. The MIT Press, Third edition, 2009, ISBN 978-0262033848

[Johnsonbaugh and Schaefer; 2010] Richard Johnsonbaugh and Marcus Schaefer: *Algorithms.* Prentice Hall, 2003, ISBN 978-0023606922

[Levitin; 2012] Anany Levitin: *Introduction to Design and Analysis of Algorithms*. Pearson, Third Edition, 2012, ISBN 978-0132316811

[McConnell; 2007] Jeffrey McConnell: *Analysis of Algorithms*. Jones & Bartlett Pub; Second edition, 2007, ISBN 978-0763707828

[Miller and Boxer; 2013] Russ Miller and Laurence Boxer: *Algorithms; Sequential and Parallel; A Unified Approach*. Cengage Learning, Third Edition, 2013, ISBN 978-1133366805

[Shackelford; 1997] Russell Shackelford: *Introduction to Computing and Algorithms.* Addison Wesley, 1997, ISBN 978-0201314519