# Backtracking algorithms

**Author(s)**
Silvio Peroni – silvio.peroni@unibo.it – https://orcid.org/0000-0003-0530-4305
Digital Humanities Advanced Research Centre (DHARC), Department of Classical Philology and Italian Studies, University of Bologna, Bologna, Italy

**Keywords**
Abstract strategy board games; AlphaGo; Backtracking; Peg solitaire

## Abstract

This chapter introduces an algorithmic technique used in constrained computational problems. Typical problems of this kind are those related to the resolution of abstract strategy board games, such as the peg solitaire. The historic hero introduced in these notes is AlphaGo, an artificial intelligence developed by Google DeepMind to play Go.

## Historic hero: AlphaGo

AlphaGo (shown in Figure 1) is an artificial intelligence developed by Google DeepMind for playing a particular board game, i.e. Go. A introduced in one of the first chapters, Go is an ancient abstract strategy board game. Due to its ample solution space, t is mainly known for being very complex to play by a computer.

**Figure 1.** The logo of AlphaGo. Source: https://en.wikipedia.org/wiki/File:Alphago_logo_Reversed.svg.

Scientists developed several artificial intelligence applications to play to Go automatically in the past. However, all of them showed their limits when tested with human expert players. Before 2016, no Go software was able to beat a human master. The approach adopted by those systems, even if sophisticated, was not enough for emulating the skills of expert human players.

In 2015, a particular department within Google declared to have developed the best artificial intelligence for playing Go and suggested testing it in an international match against one of the greatest Go players in Europe, Fan Hui. At the time of the game, Fan Hui was a professional two dan player (out of 9 dan). The match was in five sessions, and a player had to win at least three sessions out of five for winning the game. AlphaGo beat Fan Hui 5-0 and has become the first artificial intelligence to beat a professional human player. The outcomes of the match were announced in January 2016 simultaneously with the publication of the Nature article explaining the algorithm [Silver et al., 2016].

In March 2016, AlphaGo was engaged against Lee Sedol, a professional nine dan South Korean Go player, one of the best players in the world. All five sessions of the match were broadcast live in streaming video to allow people to follow the various games. AlphaGo beat Lee Sedol 4-1. Finally, in May 2017, AlphaGo was involved in a match in three sessions against the top-ranked player, the Chinese Ke Jie. Even this time, AlphaGo beat Ke Jie 3-0. After that, Google DeepMind decided to retire AlphaGo definitely from the scenes due to this match.

Several professional players have stated that AlphaGo seemed to use quite new moves if compared with the other professional players. As a consequence of its results and in several secondary games, Alpha Go gained recognition as a professional nine dan player by the Chinese Weiqi Association.

A final article about the latest evolution of the system, AlphaGo Zero, was published in Nature on the 18th of October 2017 [Silver et al., 2017]. In this article, the authors introduce the best version of AlphaGo that they have developed. The Alpha Go creators trained this new version without using any match between human champions archived in the past, as they did for training AlphaGo. In particular, they trained it against itself, and it reached a superhuman performance after only 40 days of self-training. As a result, the new AlphaGo Zero beat AlphaGo 100-0.

## Tree of choices

Usually, all the algorithms for finding a solution to abstract strategy board games use a tree, where each node represents a possible move on the board. Thus, the idea is that one comes to a particular node after executing a precise sequence of moves. Then, one can have an additional set of possible valid moves from that node to approach the solution. But, of course, to choose a particular move, one should also check if executing that move could bring a solution to our problem or it lands at a dead-end. In this latter case, one could reconsider some previous choices and, eventually, change strategy looking for some other, more promising, moves to follow.

Technically speaking, this approach is called *backtracking* in the Computational Thinking domain. In practice, backtracking algorithms try to solve a particular computational problem by identifying possible candidate solutions incrementally. Moreover, it abandons partial candidates once it is clear that they will not be able to solve the problem. The usual steps of a backtracking algorithm can be defined as follows, and consider a particular node of the tree of choices as input:

1. **[leaf-win]** if the current node is a leaf, and it represents a solution to the problem, then return the sequence of all the moves that have generated the successful situation; otherwise,
2. **[leaf-lose]** if the current node is a leaf, but it is not a solution to the problem, then return no solution to the parent node; otherwise,
3. **[recursive-step]** apply the whole approach recursively for each child of the current node until one of these recursive executions returns a solution. If none of them provides a solution, return no solution to the parent node of the current one.

In the next section, we illustrate the application of this technique for solving a particular board game: peg solitaire.

## Peg solitaire

Peg solitaire is a board game for one person only, which involves the movement of some pegs on board containing holes. Usually, in the starting situation, the board contains pegs everywhere except for the central position, which is empty. While there are different standard shapes for the board of the game, as illustrated in Figure 2, – the classic board is the English one (the number 4 in Figure 2).

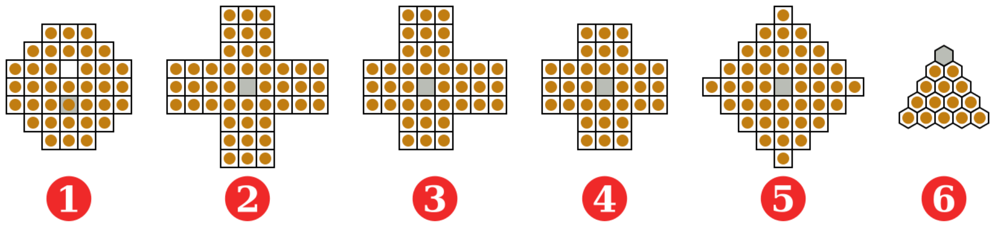

**Figure 2.** Possible standard shapes for the board of the peg solitaire – the number 4 is the English one. Figure by Julio Reis, source: https://commons.wikimedia.org/wiki/File:Peg_Solitaire_game_board_shapes.svg.

The game's goal is to come to the opposite of the starting situation: the whole board must be full of holes except the central position, which must contain a peg. It is possible to repeatedly apply the same rule to reach this goal: to move a peg orthogonally over an adjacent peg into a hole and remove the jumped peg from the board. Figure 3 illustrates an example of valid moves.

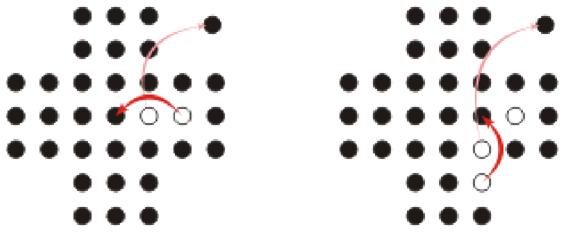

**Figure 3.** An example of two consecutive and valid moves on an English board. Source: https://ece.uwaterloo.ca/~dwharder/aads/Algorithms/Backtracking/Peg_solitaire/.

The computational problem we want to address in this chapter can be defined as follows:

**Computational problem:** find a sequence of moves to solve the peg solitaire.

A reasonable approach for finding a solution to this computational problem is based entirely on backtracking. In particular, we should develop it according to the following steps:

1. **[leaf-win]** if the last move has brought to a situation where there is only one peg, and it is in the central position, then a solution has been found, and the sequence of moves executed for coming to this solution is returned; otherwise,
2. **[leaf-lose]** if the last move has brought to a situation where there are no possible moves, then recreate the previous status of the board as if we did not execute the previous move, and return no solutions; otherwise,
3. **[recursive-step]** apply the algorithm recursively for each possible valid move executable according to the board's current status until one of these recursive executions of the algorithm returns a solution. If none of them provides a solution, recreate the previous status of the board as if we did not execute the last move and return no solutions.

The idea is to start with the initial configuration of the board as the root of a tree of moves. Then, we consider all the possible configurations reachable from such a root after a valid move, and so on. This process, in practice, would allow us to describe all the possible scenarios with quite a big tree. However, it is worth mentioning that visiting all the possible configurations is unnecessary. Instead, the algorithm can terminate with success once it reaches a final winning configuration, as summarised in Figure 4.

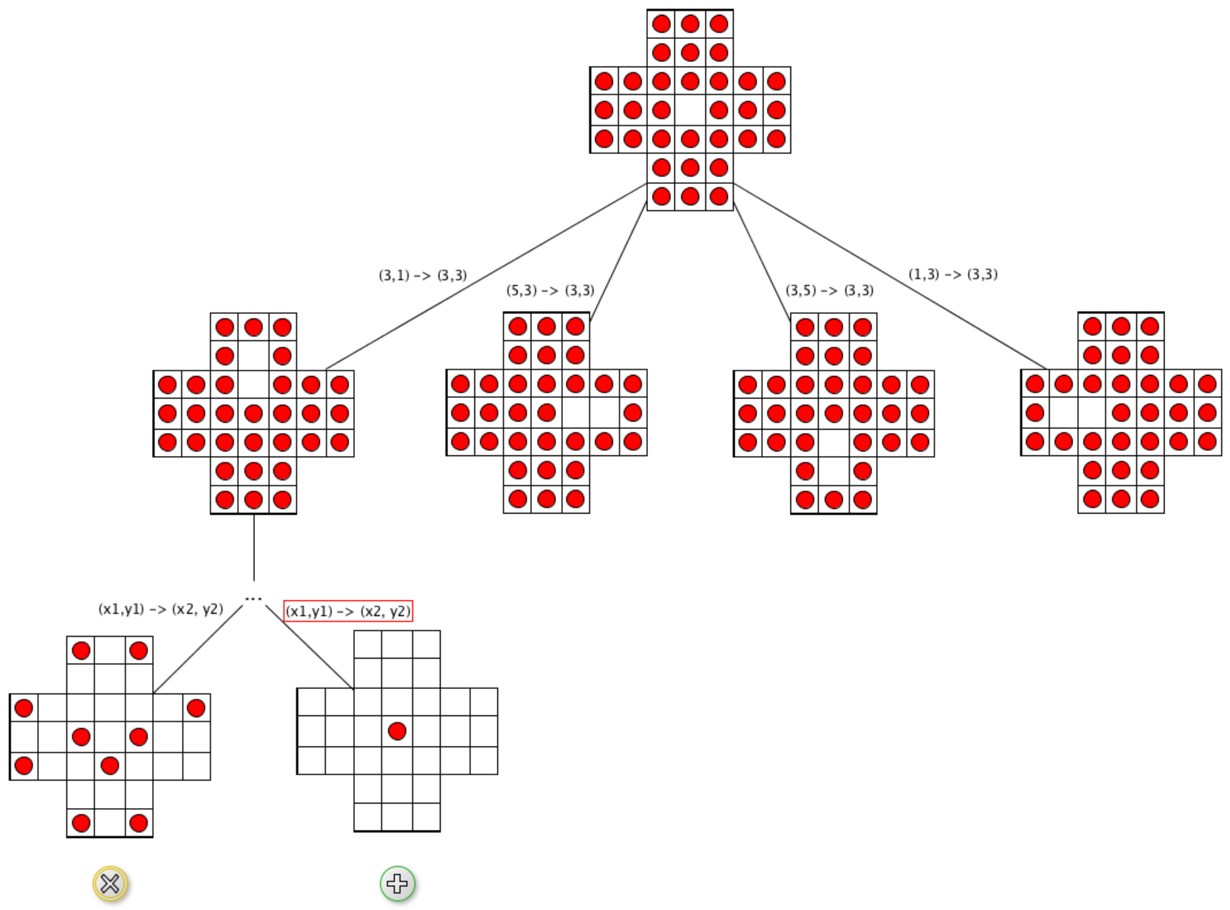

**Figure 4.** A sketch of a tree of moves a player have from the initial configuration, which is the root of the tree.

This chapter aims to develop an algorithm for finding a solution to the peg solitaire, considering an alternative board, shown in Figure 5. This board is the smallest square board on which we can obtain the complement of a given initial configuration of a board by replacing every peg with a hole and vice versa [Bell, 2007]. The advantage of using this board is that the tree of moves' dimension is relatively small compared with a classic English peg solitaire board. However, it maintains, algorithmically, all the properties of the problem of the more complex one.

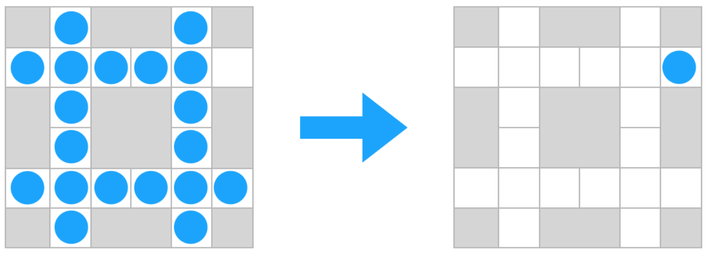

**Figure 5.** The complement problem of the peg solitaire, depicted on a 6x6 square board.

```
      (1,0)             (4,0)
(0,1) (1,1) (2,1) (3,1) (4,1) (5,1)
      (1,2)             (4,2)
      (1,3)             (4,3)
(0,4) (1,4) (2,4) (3,4) (4,4) (5,4)
      (1,5)             (4,5)
```

**Listing 1.** The representation of all the positions available in a peg solitaire with a 6x6 square board as a collection of tuples of two elements.

To implement it in Python, we need to find a way for representing all the possible positions of the peg solitaire board in a computationally sound way. To this end, we use a tuple of two elements, depicting x-axis and y-axis values, as representative of a particular position. The collection of all these tuples, shown in Listing 1, represents our board from a purely computational point of view.

```
def create_board():
    initial_hole = (5, 1)
    holes = set()
    holes.add(initial_hole)

    pegs = set([
        (1, 0), (4, 0),
        (0, 1), (1, 1), (2, 1), (3, 1), (4, 1),
        (1, 2), (4, 2),
        (1, 3), (4, 3),
        (0, 4), (1, 4), (2, 4), (3, 4), (4, 4), (5, 4),
        (1, 5), (4, 5)
    ])

    return pegs, holes
```

**Listing 2.** The function used for initialising the 6x6 square board of a peg solitaire as shown in Figure 5.

According to this organisation, we use two sets for representing a particular status of the board, i.e.:

- the set *pegs* that includes all the pegs available on the board – in the initial status, this set contains all the positions except the final one, i.e. `(3, 2)`;
- the set *holes* that includes all the positions without any peg on the board – in the initial status, this set has just one position, i.e. `(3, 2)`.

The ancillary function introduced in Listing 2, i.e. `create_board()`, provides the initial configuration of the solitaire. The result of the execution of this function is a tuple of two elements. In the first position, there is a set of all the pegs at the initial state. In the second position, there is a set of all the available holes at the initial state.

```
from anytree import Node

def valid_moves(pegs, holes):
    result = list()

    for x, y in holes:
        if (x-1, y) in pegs and (x-2, y) in pegs:
            result.append(Node({"move": (x-2, y), "in": (x, y),
                                "remove": (x-1, y)}))
        if (x+1, y) in pegs and (x+2, y) in pegs:
            result.append(Node({"move": (x+2, y), "in": (x, y),
                                "remove": (x+1, y)}))
        if (x, y-1) in pegs and (x, y-2) in pegs:
            result.append(Node({"move": (x, y-2), "in": (x, y),
                                "remove": (x, y-1)}))
        if (x, y+1) in pegs and (x, y+2) in pegs:
            result.append(Node({"move": (x, y+2), "in": (x, y),
                                "remove": (x, y+1)}))

     return result
```

**Listing 3.** The function used for retrieving all the valid moves starting from a particular configuration of the solitaire board.

Another essential task we need to deal with is taking all the possible moves one can do according to the board's current configuration. The approach used, described in Listing 3 and named `valid_moves(pegs, holes)`, try to find all the possible moves in the proximity of each hole. In particular, starting from a hole defined by the coordinates `(x, y)`, it looks for a vertical or horizontal sequence of two pegs that create the condition for performing a valid move.

Each move found is described by a particular small dictionary with the following three keys:

- *move*, which indicates the peg one wants to move;
- *in*, which shows the position where we placed the selected peg after the move (i.e. the hole in consideration);
- *remove*, which indicates the peg that we removed from the board due to the move.

As highlighted in Listing 3, we use the compact constructor for defining dictionaries on-the-fly in Python, i.e. `{<key_1>: <value_1>, <key_2>: <value_2>, ...}`. For instance, the dictionary `my_dict = {"a": 1, "b": 2}` can be obtained as well by applying the following operations: `my_dict = dict(),my_dict["a"] = 1,my_dict["b"] = 2`.

```
def apply_move(node, pegs, holes):
    move = node.name
    old_pos = move.get("move")
    new_pos = move.get("in")
    eat_pos = move.get("remove")

    pegs.remove(old_pos)
    holes.add(old_pos)

    pegs.add(new_pos)
    holes.remove(new_pos)

    pegs.remove(eat_pos)
    holes.add(eat_pos)


def undo_move(node, pegs, holes):
    move = node.name
    old_pos = move.get("move")
    new_pos = move.get("in")
    eat_pos = move.get("remove")

    pegs.add(old_pos)
    holes.remove(old_pos)

    pegs.remove(new_pos)
    holes.add(new_pos)

    pegs.add(eat_pos)
    holes.remove(eat_pos)
```

**Listing 4.** The two functions applying and undoing a particular move on the board.

We encapsulate each dictionary defining a move as the name of a tree node. In particular, we create a node using the constructor defined by the package *anytree* we introduced in the previous chapter. The function in Listing 3 returns a list of all the possible valid moves according to the particular configuration of the board.

We need two additional functions. They take as input a move defined by a tree node and a particular configuration of the board and returns the new configuration as if we apply or undo the move, respectively. These functions, i.e. `apply_move(node, pegs, holes)` and `undo_move(node, pegs, holes)`, are defined in Listing 4.

```
def solve(pegs, holes, last_move):
    result = None

    if len(pegs) == 1 and (5, 1) in pegs:  # leaf-win base case
        result = last_move
    else:
        last_move.children = valid_moves(pegs, holes)

        if len(last_move.children) == 0:  # leaf-lose base case
            undo_move(last_move, pegs, holes)  # backtracking
        else:  # recursive step
            possible_moves = deque(last_move.children)

            while result is None and len(possible_moves) > 0:
                current_move = possible_moves.pop()
                apply_move(current_move, pegs, holes)
                result = solve(pegs, holes, current_move)

            if result is None:
                undo_move(last_move, pegs, holes)  # backtracking

    return result
```

**Listing 5.** The final function for looking for a sequence of moves that depicts a solution to the peg solitaire computational problem. The source code of this listing and all the previous ones is available as part of the course material and includes the related test cases.

Finally, we need to use a particular comparison operator, i.e. `is` (and its inverse, `is not`), which we did not use in the past chapter. This operator checks the **identity** of the objects instead of the values they may refer to. In particular, `<obj1> is <obj2>` returns *True* if the two objects (or two variables referring to two objects) are the **same** object; otherwise, it returns *False*. The `is not` operator works oppositely. It is worth mentioning that checking for the identity of two objects is different from checking for their equality (through the operator `==`). In particular, consider the following two lists stored in different variables:

```
list_one = list()
list_one.append(1)
list_one.append(2)
list_one.append(3)

list_two = list()
```

```
list_two.append(1)
list_two.append(2)
list_two.append(3)
```

The expression `list_one == list_two` returns *True* since the lists are equal according to the values they contain. However, the expression `list_one is list_two` returns *False* since they are two distinct instances of the Python class *list*.

We have now all the ingredients for implementing the function for finding a solution of the peg solitaire, according to the backtracking principles introduced at the beginning of this section. Listing 5 presents the final function. For running the function properly, we should initialise the `last_move` parameter as the root node of the tree of moves built by the function's execution, e.g. `Node("start")`.

## Exercises

1. Propose some variation to the implementation of the peg solitaire exercise to make it more efficient – in particular, avoiding unsuccessful configurations if they have been previously encountered while looking for a solution.
2. We define a labyrinth as a set of tuples representing the various cells of the paths within the labyrinth. The tuples are organised in an x/y grid, similar to the way used in Listing 1 for the peg solitaire, such as the one proposed as follows:

   ```
          (1,0)         (3,0) (4,0) (5,0)
   (0,1)  (1,1)  (2,1)  (3,1)       (5,1)
   (0,2)         (2,2)        (4,2) (5,2)
   (0,3)         (2,3)  (3,3)       (5,3)
   (0,4)                      (4,4)
   (0,5)  (1,5)  (2,5)  (3,5) (4,5)
   ```

   Write the function `solve_labyrinth(paths, entrance, exit, last_move)`, which takes as input the paths of the labyrinth (such as the ones mentioned above), two tuples representing the entrance and the exit of the labyrinth, and the last move did. The function returns the last move done to reach the exit if the labyrinth has an escape; otherwise, it returns `None`. Accompany the implementation of the function with the appropriate test cases.

## Acknowledgements

The author wants to thank one of the students of the Digital Humanities and Digital Knowledge second-cycle degree at the University of Bologna – Tanise Pagnan Ceron and Francesca Borriello – for suggesting corrections and improvements to the text of this chapter.

## References

Bell, G. I. (2007). A Fresh Look at Peg Solitaire. Mathematics Magazine, 80 (1): 6-28. DOI https://doi.org/10.1080/0025570X.2007.11953446, freely available at https://www.jstor.org/stable/pdf/27642987.pdf

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V. (2016). Mastering the game of Go with deep neural networks and tree search. Nature, 529 (7587): 484-489. DOI: https://doi.org/10.1038/nature16961, freely available at http://web.iitd.ac.in/~sumeet/Silver16.pdf

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I, Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., Hassabis, D. (2017). Mastering the game of Go without human knowledge. Nature, 550 (7676): 354-359. DOI: https://doi.org/10.1038/nature24270, freely available at https://www.gwern.net/docs/rl/2017-silver.pdf