Backtracking algorithms

Author(s)
Silvio Peroni – silvio.peroni@unibo.it
Department of Classical Philology and Italian Studies, University of Bologna, Bologna, Italy

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

Abstract

These lecture notes introduce an algorithmic technique that is usually adopted in constrained computational problems such as those ones 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 for playing the game of Go.

Historic hero: AlphaGo

AlphaGo (its logo is shown in Figure 1) is an artificial intelligence developed by Google DeepMind for playing a particular board game, i.e. Go. As already introduced in one of the first lectures, Go is a very ancient abstract strategy board game, which particularly known for being very complex to play by a computer, due to its large solution space.

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

Several artificial intelligence applications have been developed in the past in order to play to Go automatically. However, all of them showed their limits when tested with human expert players of the game. In fact, no Go software was able to beat a human master, since the approach adopted by those systems, even if sophisticated, was not enough for emulating the actual 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 to test it in an international match against one of the most strongest Go players in Europe, Fan Hui. At the time of the game, Fan Hui was a professional 2 dan player (out of 9 dan). The match was organised in five sessions, and a player had to win at least three sessions out of five for winning the match. AlphaGo beat Fan Hui 5-0, and has become the first artificial intelligence to beat a professional human player of the game. 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 9 dan South Korean Go player, one of the best player in the world. All the five sessions of the match were broadcasted live in streaming video, so as 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 of the game, the Chinese Ke Jie. Even this time, AlphaGo beat Ke Jie 3-0. As a consequence of this match, Google DeepMind decided to retire AlphaGo definitely from the scenes.

Several professional players have stated that AlphaGo seemed to use quite original moves if compared with the other professional players. In addition, as a consequence of the results AlphaGo gained in the aforementioned matches, as well as in several secondary games, it was formally recognised as a professional 9 dan player by Chinese Weiqi Association.

A final article about the latest evolution of the system, AlphaGo Zero, has been published in Nature the 18th of October 2017 [Silver et al., 2017]. In this article, the authors introduce the best version of AlphaGo that they have developed, which has been trained without using any existing human knowledge – i.e. the matches between human champions that have been archived in the past, that have been actually used for training AlphaGo. In fact, it was trained against itself, and it reached a superhuman performance after only 40 days of self-training. The new AlphaGo Zero beat AlphaGo 100-0.

Tree of choices

Usually, all the algorithms for finding a solution to abstract strategy board games are based on a tree, where each node represents a possible move that can be done on the board. Thus, the idea is that one comes to a particular node after having executed a precise sequence of moves, and from that node, one can have available an additional set of possible valid moves to perform

so as to approach the solution. Of course, in order to choose to follow a particular move, i.e. to step into a particular child node of the tree, one could also check if executing that move could bring to a solution to our problem or it lands to a dead-end. In this latter case, one could re-consider some previous choices and, eventually, could change strategy looking for some other, more promising, moves to follow.

Technically speaking, in the Computational Thinking domain, this approach is called *backtracking*. In practice, backtracking algorithms try to find a solution to a particular computational problem by identifying possible candidate solutions incrementally, and it abandons partial candidates once it is clear that they won't be able to provide a solution to 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 back the parent node; otherwise,
3. **[recursive-step]** apply recursively the whole approach 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 back the parent node of the current one.

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

Peg solitaire

The peg solitaire is a board game for one person only which involves the movement of some pegs on board containing holes. Usually, the starting situation has the entire board is filled up with pegs 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).

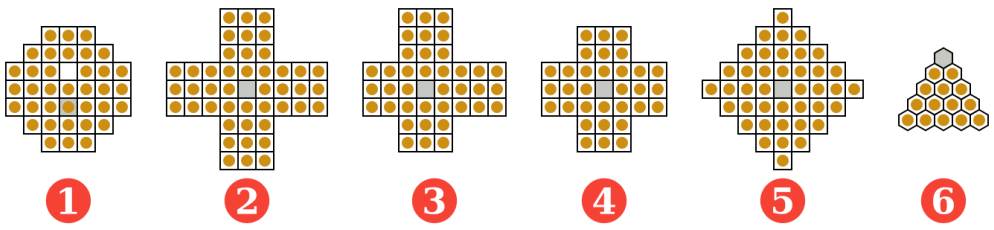

Figure 2. Possible standard shapes for the board of the peg solitaire – the number 4 is the English one. Figure by Julio Reis, source: https://commons.wikimedia.org/wiki/File:Peg_Solitaire_game_board_shapes.svg.

The goal of the game is to come to the opposite of the starting situation, where the whole board is full of holes except the central position which must contain a peg. This is possible by applying repeatedly the same rule: moving orthogonally a peg over an adjacent peg into a hole two position away, removing the jumped peg from the board. An example of valid moves is illustrated in Figure 3.

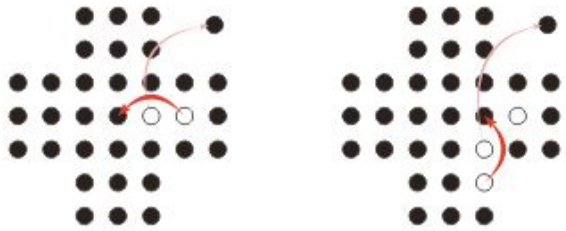

Figure 3. An example of two consecutive valid moves on an English board. Source: https://ece.uwaterloo.ca/~dwharder/aads/Algorithms/Backtracking/Peg_solitaire/.

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

Computational problem: find a sequence of moves that allows one to solve the peg solitaire.

A reasonable approach for finding a solution to this computational problem is based entirely on backtracking. In particular, it should be organised 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 positioned 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 the last move was not executed at all, and return no solutions; otherwise,
3. **[recursive-step]** apply recursively the algorithm for each possible valid move executable according to the current status of the board, 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 the last move was not executed at all, and return no solutions.

In practice, the idea is to start with the original configuration of the board as the root of a tree of moves, and then considering all the possible configurations reachable from the root after a valid move as its children, and so on. This, in practice, would allow us to describe all the possible scenarios with a quite big tree. It is worth mentioning that it is not necessary to visit all the possible configurations, but rather the algorithm can terminate successfully once the final winning configuration is actually reached. This is graphically summarised in Figure 4.

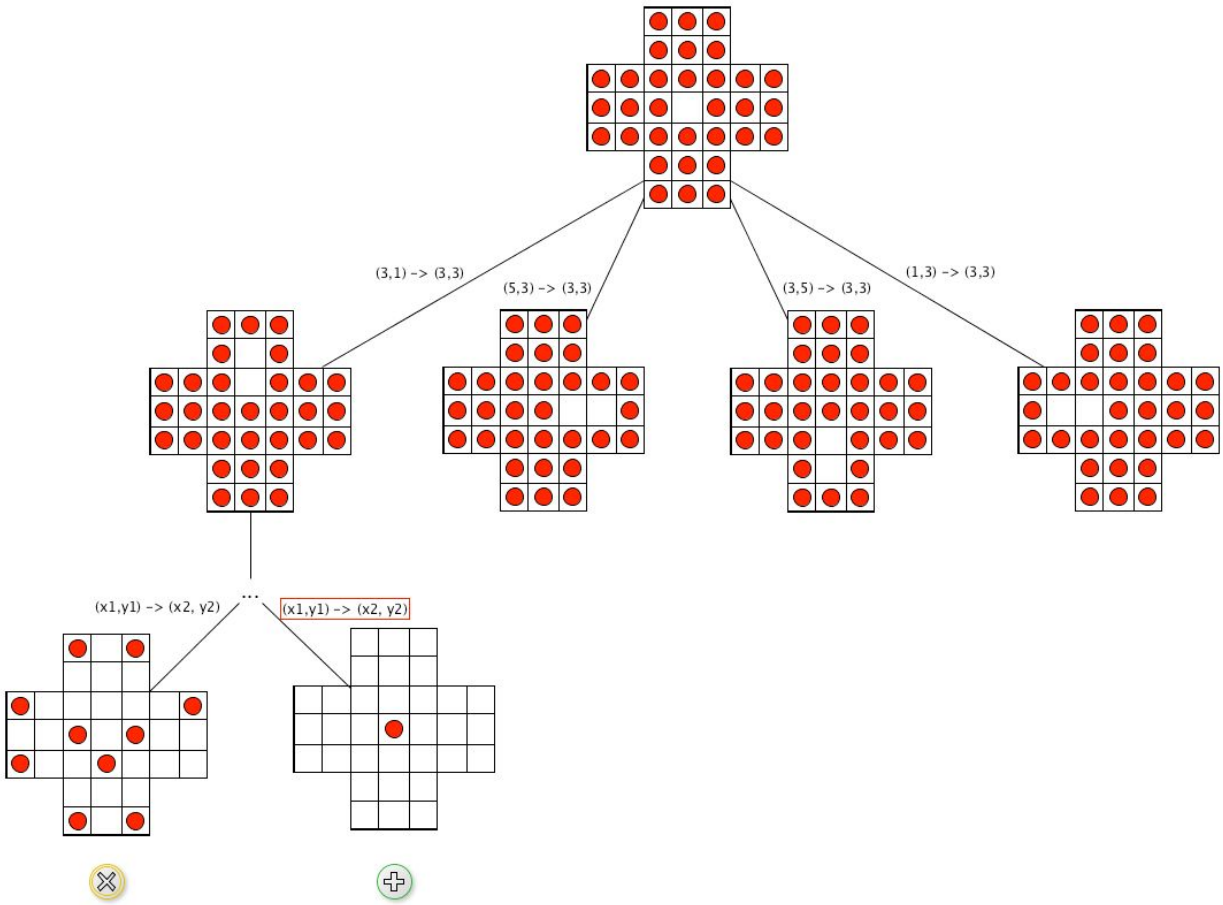

Figure 4. A sketch of the possible tree of moves that a player can done starting from the initial configuration, which is the root of the tree.

The goal of this lecture is to develop an algorithm for finding a solution to the peg solitaire, considering an alternative board, i.e. the 6x6 square board. This board is the smallest square board on which the *complement problem*, i.e. obtaining the complement of a given an initial configuration of a board by replacing every peg by a hole and vice versa, as depicted in Figure 5. The advantage of using this board is that the dimension of the tree of moves is rather small if compared with the one of a classic English peg solitaire board, while it maintains, algorithmically, all the properties of the problem of the more complex one.

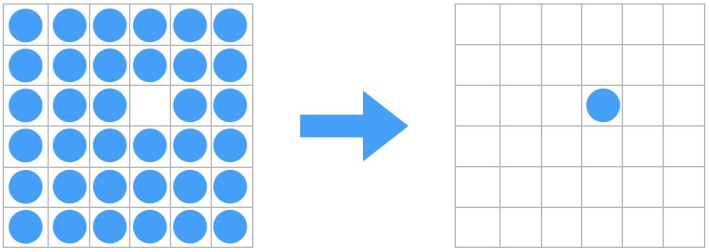

Figure 5. The complement problem of the peg solitaire, depicted on a 6x6 square board.

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

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

The first thing we have to take into account for implementing the aforementioned rules in Python is to find a way for representing all the possible position of the peg solitaire board in a way that is computationally sound. To this end, we use a tuple of two elements, depicting x-axis and y-axis values, as representative of a certain position. The collection of all these tuples, shown in Listing 1, represents our board from a purely computational point of view.

```
from itertools import product

def create_6x6_square_board():
    initial_hole = (3, 2)
    holes = set()
    holes.add(initial_hole)

    pegs = set()
    cell = range(6)
    # The 'product' function does a cartesian
    # product between the values of the two
    # ordered collections specified as input
    pegs.update(product(cell, cell))
    pegs.remove(initial_hole)

    return pegs, holes
```

Listing 2. The function used for initialising the 6x6 square board of a peg solitaire. The function `product` is defined in the Python package `itertools` and returns the Cartesian product of two or more collections of objects. For instance, `product([0, 1], ["a", "b", "c"])` returns a collection composed by the following tuples: `(0, "a"), (0, "b"), (0, "c"), (1, "a"), (1, "b"), (1, "c").`

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 includes all the position 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 includes just one position, i.e. `(3, 2)`.

The initial configuration of the solitaire is provided by the particular ancillary function introduced in Listing 2, i.e. `create_6x6_square_board()`. The result of the execution of this function is a tuple of two elements, wherein the first position there is the set of all the pegs at the initial state, while in the second position there is the 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 algorithm used for retrieving all the valid moves starting from a particular configuration of the solitaire board.

Another important algorithm we would like to have concerns to take all the possible moves one can do according to the current configuration of the board defined by the sets *pegs* and *holes*. 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 indicates the position where the selected peg is placed after the move (i.e. the hole in consideration);

- *remove*, which indicates the peg that is removed from the board as consequence of the move.

```
def apply_move(node, pegs, holes):
    move = node.name
    old_pos = move["move"]
    new_pos = move["in"]
    eat_pos = move["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["move"]
    new_pos = move["in"]
    eat_pos = move["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 responsible for applying and undoing a particular move on the board.

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`.

Each dictionary defining a move is then encapsulated as the name of a tree node. In particular, every node is created by using the constructor defined by the package *anytree* we have

introduced in the previous lecture. The algorithm in Listing 3 returns a list of all the possible valid moves according to the particular configuration of the board.

Finally, we need two additional algorithm that takes a move defined by a tree node and a particular configuration of the board as input, and returns the new configuration as if the move is applied or if the move is undone respectively. These algorithms, i.e. `apply_move(node, pegs, holes)` and `undo_move(node, pegs, holes)`, are defined in Listing 4.

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

    if len(pegs) == 1 and (3, 2) 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 algorithm 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 material of the course, and includes also the related test cases.

We have now all the ingredients for implementing the actual algorithm for finding a solution of the peg solitaire, according to the backtracking principles introduced at the beginning of this section. The final algorithm is introduced in Listing 5.

Exercises

1. Propose some variation to the implementation of the peg solitaire exercise in order to make it more efficient – in particular, avoiding unsuccessful configurations if they have been already encountered previously while looking for a solution.

References

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