Monte Carlo Tree Search: A Review

Peter Racioppo

June 19, 2020

Abstract

Monte Carlo Tree Search is a recursive search algorithm used in decision processes, typically to solve problems which can be posed as games. This paper reviews the important concepts in the algorithm and discusses the implementation of a Monte Carlo method for Tic-Tac-Toe game-playing.

1 Introduction

Decision processes, in which a sequence of actions much be chosen to maximize an objective function, effectively model problems in a wide variety of fields, including economics, biology, and many fields of engineering [1]. Important approaches to these problems have included graph search algorithms, game theoretic approaches, and logic-based approaches developed by the artificial intelligence community, statistical and probabilistic algorithms from machine learning, and control theoretic methods based on dynamical systems theory and stochastic estimation. Many decision processes can be modeled as n-player games, in which n agents take turns making moves, each attempting to maximize an objective function. In this setting, a decision tree can be used to enumerate the possible moves at each step in the decision process. Good moves can then be found by simulating games and performing a probabilistic search of the decision tree. Monte Carlo Tree Search (MCTS), one such search algorithm, proceeds by recursively updating the search tree to focus on more promising regions of the state space. Since its introduction in 2006 for playing the strategy game Go [2], MCTS has been widely applied as an important tool in game AIs. Google's AlphaGo program, which employed MCTS along with deep learning and reinforcement learning, famously defeated Go world champion Lee Sedol in 2016 [3], and has also been successfully employed in incomplete-information games such as poker, online strategy games, and more recently, the general game-playing algorithm AlphaZero [2,4-6].

2 Approaches to Solving Decision Problems

In optimal control, the calculus of variations and dynamic programming techniques are used to find a control law, a time-dependent input to a system of differential equations, which takes the state of a system from an initial condition to a desired final condition, while attempting to minimize a cost function [7]. For example, in the special case of a quadratic cost function of the states and inputs, the optimal solution can be found in closed form as the Linear Quadratic Regulator [8]. These techniques are limited to systems that can be tractably modeled as system of differential equations or finite difference equations.

A more general model of a discrete-time stochastic control process is the Markov Decision Process (MDP). MDPs are an extension of Markov chains in which, at each time step, a control algorithm may take an action a which influences the probability that the current state s will transition to some other state in the state space S. A Markov decision process is a four-tuple (S, A, P_a, R_a), where S is a

set of states, A is a set of possible actions, $P_a(s, s')$ is the probability of moving from state s to s', and $R_a(s, s')$ is the reward for moving from state s to s'. The state transition probability P_a satisfies the Markov property: it is conditionally independent of all previous states and actions, given the current state and action. MDPs are typically solved using dynamic programming techniques or reinforcement learning [9-11].

Logic-based approaches have been a central feature of many artificial intelligence algorithms. Algorithms based on formal logic can perform high-level reasoning, readily understood by human designers. However, formal logic models require that we specify rules for every possible scenario, and don't allow for logical variables to be changed as new information becomes available. Probabilistic programming attempts to bridge the gap between probabilistic modeling and formal logical methods by, for instance, converting statistical models to weighted Boolean formulas, stored in binary decision trees, and then using algorithms for weighted Boolean model counting [12-14]. Inference in probabilistic programming is often performed using Markov Chain Monte Carlo to sample from complex probability distributions arising from the use of many random logical variables [15-20].

Game theoretic approaches are natural for systems which can be modeled as discrete agents making discrete decisions, as for example, in two-player games like Checkers, Chess, Go, or Tic-Tac-Toe. The values of particular states are determined by a position evaluation function. In simple games with few possible moves and short game lengths, such as Tic-Tac-Toe, a natural evaluation function is simply to reward a "win" at the end of the game and penalize a "loss." In more complicated games, in which the outcome of the game takes place after very long sequences of moves, it may be necessary to assign evaluation function values to intermediate states, to represent when a position is advantageous or confers an increased probability of winning.

A minimax algorithm [21] is an algorithm for choosing from a finite set of moves in a game against competing players. At each move, a player using the minimax algorithm makes the move which maximizes the evaluation function, assuming the other player will make the moves which minimize the objective function. In a simple game like Tic-Tac-Toe, a decision tree can be iteratively constructed from any given position until every possible ending of the game. Recursively moving from the leaves of the tree to the root, enumerating the possibilities if the player always attempts to maximize the objective function and the opponent always attempts to minimize the objective function, the globally optimal solution can be determined. Let w be the width of the tree, the number of available moves at each turn, and let d be the depth of the tree, the number of turns before the game ends. In a game with fixed w and d, the game search tree has w^d nodes. Thus, in games with large w or d, an exhaustive search procedure becomes computationally intractable.

Alpha-beta pruning allows minimax to be applied in larger search trees by pruning branches of the tree that will never be played. We begin by exploring the tree in a depth-first manner. Proceeding back along this path, we then prune branches which would have resulted in a worse outcome for either player. For example, if we are playing chess, and discover that a subtree will lead to a checkmate in two moves (while other options do not), we can remove this subtree from consideration. This procedure is implemented by continuously updating two values, alpha and beta, which record, respectively, the minimum score that the maximizing player can obtain and the maximum score that the minimizing player can obtain. If a move causes alpha to be greater than beta, it can be removed from consideration. The efficiency of alpha-beta pruning can be improved with a variety of heuristics, including searching over narrow windows of the decision tree. The killer heuristic, for example, attempts to increase the probability of a cutoff, that is the removal of a subtree, by first considering moves that have already been evaluated to be good at the same depth in another subtree. [1,22]

The nodes of a decision tree can be explored with simple algorithms such as breath first search, which explores nodes near the root of the tree before moving to deeper layers, or depth first search, which explores the tree as deeply as possible before returning to higher layers [23]. The SSS* algorithm, introduced in 1979, searches a decision tree in a best-first manner, at each step expanding from the most-promising node, according to some search heuristic [24]. The best-first approach of SSS* can be compared to the A* algorithm for motion planning in a graph, which at each step estimates the cheapest path from a starting point [1,25], and rapidly-exploring random trees (RRTs), for search in continuous spaces [26]. Iterative deepening depth-first search performs depth-first searches only up to a certain depth, and iteratively deepens in a best-first manner. [27]

When the number of possible moves in a game is very large, randomized search methods may help address the difficulties faced by game theoretic algorithms. Note that this approach may be less effective if the set of possible moves is very large, as is the case for example in the optimal control setting, when the states are continuous. When exploring a decision tree, if we discover that a subset of the possible moves are more commonly leading to positive outcomes in simulation, we can concentrate our search on this subset. Recursively narrowing the field of search using feedback from the results of simulated games is precisely the strategy of Monte Carlo Tree Search.

3 Overview of the Algorithm

The simplest version of MCTS is to play many random games, with no bias toward any of the possible moves. We then simply pick the move which resulted in the greatest proportion of wins. More sophisticated versions of the algorithm use feedback to narrow the field of search. The steps of MCTS can be roughly broken into four parts:

Selection: Starting from the root R of the tree, move through the tree until reaching a leaf node L, that is a node from which a game has not already been played, and play a game from this starting point.

Expansion: Choose a node C, one of the children of L, from which to simulate random games.

Simulation: Simulate a random game starting from C, a so-called playout. The simplest method of simulating a game is to randomly sample each move from a uniform distribution, but alternatives include incorporating neural networks and playing against previous versions of the algorithm.

Back-propagation: Update the nodes on the path from C to R depending on the result of the playout.

At each step, the algorithm must balance the tradeoff between exploitation of the paths which have already been explored and exploration of new regions of the state space. Modern MCTS incorporates the concept of the upper confidence bound from the multi-armed bandit problem, which is traditionally posed as the following: imagine that we are playing on a number of slot machines, each with some fixed probability of winning. We would like to maximize our total number of wins, but to do so we must trade off between the need to find the machine with the best odds and to exploit the machine which we have already estimated to be best. Thus, our goal is to select the $A_t = \text{argmax}(q^*(a))$, where $q^*(a)$ is the reward distribution. Given an estimate $Q_t(a)$ for $q^*(a)$, we can simply choose $A_t = \text{argmax}(Q_t(a))$, but this greedy choice doesn't account for the need to perform further exploration to update our estimate. We define the regret as the expected loss due to not always playing the best machine. Lai and Robbins showed in 1985 that the regret grows at least at a logarithmic rate as a function of the number of plays [28]. Auer et al. showed in 2002 that a logarithmic growth rate for the regret can be achieved with a simple and fast algorithm for sampling from the possible machines, when the reward distribution for the payoffs of the machines

is an arbitrary function with bounded support [29]. In the multi-armed bandit problem, we want the difference between the actual reward distribution $q^*(a)$ and our estimated distribution $Q_t(a)$ to be small. We can bound this difference with Hoeffding's inequality: $\Pr(|Q_t(a) - q^*(a)| > U_t(a)) \leq e^{-2N_t U_t(a)^2}$, where N_t is the sample size at iteration t and U_t is a threshold. Suppose this probability is equal to some small value l. Inverting the expression, we have $U_t(a) = \sqrt{-\ln(\frac{l}{2N_t(a)})}$. We can then define an upper confidence bound policy as $UCB(a) = \text{argmax}(Q_t(a) + \sqrt{-\ln(\frac{l}{2N_t(a)})})$ and we can set l to follow some annealing schedule as a function of the number of iterations, for instance $l = t^{-4}$. The UCT bound (Upper confidence bound 1 applied to trees) applied this heuristic in the context of decision trees in 2006 [30]. Like in the multi-armed bandit problem, we attempt to balance exploitation and exploration by selecting, at each step, the node which maximizes $\frac{w_i}{n_i} + c\sqrt{\frac{\ln(N_i)}{n_i}}$, where w_i is the number of wins for the node, starting from the ith move, n_i is the total number of simulations for the node, starting from the ith move, N_i is the total number of simulations for the parent of the node, starting from the ith move, and c is an exploration parameter, typically learned as a hyperparameter.

4 History of MCTS

Monte Carlo techniques relying on random sampling date to applications in statistical physics in the 1940s. The use of these techniques for decision making in games can be traced to B. Abramson's use in 1987 of a Monte Carlo search together with a minimax algorithm to evaluate an "expected-outcome model. [31]" In 1992, B. Brügmann applied these techniques to the game Go [32], a perfect information game which has long been considered a major challenge among game AI researchers because of its large search space [33].

In 2005, Chang et al. introduced the Adaptive Multi-stage Sampling (AMS) algorithm, a recursive Monte Carlo search method based on Thompson sampling, and intended for modeling Markov decision processes [34]. The AMS algorithm performs sampling in order to approximate the optimal solution of a Markov Decision Process, with a bias which converges to zero in the number of steps.

These results were extended to decision tree searches for games in 2006 [35,36], and the upper confidence bound rules for balancing exploration and exploitation in search trees were described and implemented for Go in the same year [37,38]. AlphaGo, developed by Google's DeepMind team, improved on previous MCTS algorithms by incorporating reinforcement learning and recent advances in deep learning for policy selection and evaluation [39], and this program went on to defeat human experts in Go matches in 2015 and 2016. Reinforcement learning was used to train a policy network by employing self-play [33].

MCTS has been shown to converge to optimal play as the number of iterations goes to infinity in board-filling games such as Go and Chess [40]. Recent work by Shah, Xie, and Xu in 2020 established bounds for the multi-arm bandit problem, without the assumption of stationarity [41]. The authors show that introducing a polynomial rather than logarithmic term in UCB is sufficient for MCTS to approximate the reward distribution for a non-stationary multi-armed bandit problem. Other recent work has highlighted a number of similarities between Monte Carlo Tree Search and reinforcement learning, including their evaluation of states, and demonstrated how the application of RL semantics to tree search can result in a large number of new MCTS-like algorithms [42]. New applications such as "narrative generation" in story-telling and language generation are also being explored [43].

5 Implementation of a Monte Carlo Tic-Tac-Toe Algorithm

In this section, we discuss an implementation of a pure Monte Carlo game search for Tic-tac-toe, implemented in Python. We first discuss basic functionality for playing a random game, and then discuss the functions used in the Monte Carlo search method.

The function *f_Rand_TTT(n)* plays a random game of Tic-Tac-Toe on an $n \times n$ board between players X and O. We assume that X always goes first. At each step, the array pos holds the list of remaining possible moves. After a move is selected, we remove it from pos and update the array v, which keeps track of what pieces have been played at each position on the board: 1 for X, -1 for O, and 0 for an empty square. At each step, v is reshaped to a matrix M, which enables us to quickly check whether a game has been won. If either player occupies an entire row, an entire column, the main diagonal, or the off diagonal of M, that player wins. An indicator variable $Winner$, initialized to zero, is used to indicate the winner: 1 for X, -1 for O, and 0 for a tie. The function outputs M and $Winner$, which are input to *f_Plot_board(M,Winner)*, a function that prints the name of the winner and plots the pieces on the board. An example game is shown in Fig. 1. The function *f_Play_N_Random_Games(N,n)* plays N random games and calculates the percent won, lost, and tied. In 1,000 matches on a 3 by 3 board, the first player (X) won 58.49% of matches, the second player (O) won 28.55% of matches, and 12.96% of the matches resulted in a draw.

We now implement a pure Monte Carlo search algorithm that considers only the next move and then plays random games starting from each possible next move. The random games are played with replacement. For each move, we compute the total number of wins minus the total number of losses in our random games. We then pick the move that maximizes the number of wins minus the number of losses. This process amounts to maximizing an empirical estimate of the expectation of winning after playing that move.

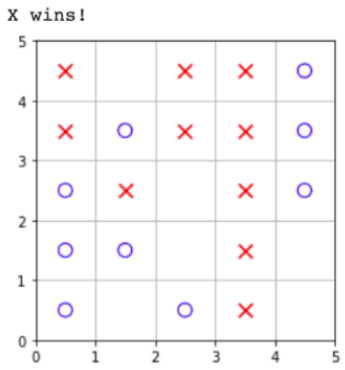


Figure 1: A random Tic-tac-toe game on a 5 by 5 grid.

The function *f_Rand_i(n,x_moves,o_moves)* plays a random game on an $n \times n$ board, starting from a position specified by *x_moves*, *o_moves*, the positions where Xs and Os have already been played. We begin by removing the set of played moves $prev\_moves = \{x\_moves, o\_moves\}$ from the set of possible moves pos at the beginning of the game, and we store the remaining possible moves in the array pos_i. The game then proceeds as in *f_Rand_TTT(n)*. The function *f_Append_Test_Move(x_moves,o_moves,test_move,n)* appends a test move to the list of moves that have already been made by X or O, depending on whose turn it is. The function *f_Branch_from_test_move(x_moves,o_moves,test_move,n,it)* plays it random games starting from the position *test_move* and counts the total number of wins in this simulation. Since there are only two players, we can count the balance of wins by simply adding and subtracting from the same counter variable. This function first calls *f_Append_Test_Move(x_moves,o_moves,test_move,n)* and then calls *f_Rand_i(n,x_moves,o_moves)* to play from this position. The function *f_Monte_Carlo_Outcomes(pos,x_moves,o_moves,n,it)* takes as input the moves already made by X and O, and lists the set of all pos-

sible moves. The function then calls *f_Branch_from_test_move(x_moves,o_moves,test_move,n,it)* to simulate games starting at each test move in this set, and returns the number of wins minus the number of losses for each test move. Here, the variable it determines the number of Monte Carlo simulations used in each simulation of a test move.

As a test of these functions, *f_Branch_from_test_move(x_moves,o_moves,test_move,n,it)* was used to determine the first move on a blank board. In 100 trials, with $it = 100$, X chose the center position as its first move in 89% of the matches, while with $it = 10$ it chose the center position in ony 64% of the matches. This confirms that the algorithm is playing more intelligently than random, since the center position is intuitively the best first move. However, it also speaks to the weakness of this first attempt at a random method, since the the AI should obviously always choose the center position. The function *f_Monte_Carlo_TTT(X_mode,O_mode,n,it)* plays a game of Tic-tac-toe on an $n \times n$ board, with X/O playing using the Monte Carlo algorithm if *X_mode / O_mode* = 1 and randomly if *X_mode / O_mode* = 0. The function *f_Play_N_Games(X_mode,O_mode,n,it,N)* plays N games on an $n \times n$ board, and counts the number of wins and losses for X and O. In $N = 100$ games with $it = 100$, when *X_mode* = 1 and *O_mode* = 0, X won 91% of matches, O won 7% and 2% were draws. Decreasing the number of iterations per test move to $it = 10$ decreased performance only slightly, with X winning 88.0%, O winnng 12.0%, and 0.0% ending in draws. In $N = 100$ games with $it = 100$, when *X_mode* = 0 and *O_mode* = 1, X won 2% of matches, O won 74%, and 24% were draws. In 100 games with both players using the Monte Carlo algorithm, X won 49.0%, O won 40.0%, and 11.0% ended in draws.

To improve the pure Monte Carlo search which we have so far implemented, we can attempt to recursively focus the search on more promising sections of the decision tree. One simple method to do this is to simply repeat the search on subsets of the tree which return the best results in the previous iteration. In particular, if there are n possible moves, we identify the best half of these and repeat the search, then repeat on the best half of this subset, and so on. This involves $O(\ln_2(n))$ Monte Carlo search phases. The function *f_Exp_Window_Monte_Carlo_TTT* is the same as *f_Monte_Carlo_TTT*, except that after the first Monte Carlo search on all possible moves, we repeat the search on the four best moves, and then again on the two best moves. Surprisingly, although the implementation appears to be correct, this method seems to marginally *worsen* the performance of the algorithm. I believe this may be a side-effect of the fact that our Monte Carlo sampler samples with replacement, and this effect becomes more detrimental as the number of possible moves decreases. In any case, sampling with replacement should intuitively improve the results.

6 Conclusion

I intend to next implement a minimax algorithm, and then implement a complete MCTS using the UCT method and backpropagation. Even for Tic-tac-toe, MCTS is a computationally expensive algorithm, and efficient use of data structures (e.g. a cache to hold visited nodes) is a necessity. Furthermore, for Tic-tac-toe, we can take advantage of rotational symmetries to considerably reduce the number of games which must be simulated. Recursive search algorithms like MCTS, especially when combined with powerful techniques from deep learning and reinforcement learning, are a promising class of algorithms. Extending their applications to richer classes of problems which can be modeled by Markov decision processes, and perhaps combining them with probabilistic programming or control theoretic methods, appear to be promising areas of research.

7 References

[1] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach (2nd ed.)*. 2003. Upper Saddle

River, New Jersey: Prentice Hall, pp. 163–171, ISBN 0-13-790395-2.

[2] D. Silver, et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm.* 2017.

[3] *Artificial intelligence: Google's AlphaGo beats Go master Lee Se-dol.* BBC News. 2016.

[4] J. Rubin and I. Watson. *Computer poker: A review.* 2011. Artificial Intelligence.

[5] *Monte-Carlo Tree Search in TOTAL WAR: ROME II's Campaign AI.* AI Game Dev.

[6] D. Silver, et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm.* 2017. arXiv:1712.01815.

[7] J. Speyer, D. Jacobson. *Primer on Optimal Control Theory.* 2010. Advances in Design and Control. PricewaterhouseCoopers LLP.

[8] P, Antsaklis, A. Michel. *Linear Systems.* 2006. Birkhauser Boston.

[9] R. Bellman. *A Markovian Decision Process.* 1957. Journal of Mathematics and Mechanics.

[10] R. Bellman. (2003) [1957]. *Dynamic Programming*. Princeton, NJ: Princeton University Press.

[11] D. Bertsekas. *Dynamic Programming and Optimal Control.* 1995.

[12] D. Fierens, et al. *Inference and learning in probabilistic logic programs using weighted Boolean formulas*.

[13] A. Kimmig, et al. *Theory and Practice of Logic Programming*. 2015.

[14] Valiant, Leslie G.. *The Complexity of Enumeration and Reliability Problems.* SIAM J. Comput. 8 (1979): 410-421.

[15] S. Zhu and Wu, Y. Wu. *Computer Vision: Statistical Models for Marr's Paradigm.* 2020.

[16] Gelman et al. *Bayesian Data Analysis*. (pg. 290-291). Chapman & Hall, 2004.

[17] Y. Wu. *A Note on Monte Carlo Methods. UCLA Statistics*. STATS 102C, 2017.

[18] A. Barbu and S. Zhu. *Monte Carlo Methods*. Springer Nature Singapore Pte Ltd. 2020.

[19] A. Sinclair. *CS294 Markov Chain Monte Carlo: Foundations & Applications*. 2009. (*https://people.eecs.berkeley.edu/ sinclair/cs294/n2.pdf*).

[20] C. Robert. *The Metropolis–Hastings algorithm*. Universite Paris-Dauphine, University of Warwick, and CREST. 2016. (*https://arxiv.org/pdf/1504.01896.pdf*)

[21] M. Maschler, E. Solan, S. Zamir. *Game Theory.* 2013. Cambridge University Press. pp. 176–180. ISBN 9781107005488.

[22] D. Edwards, T. Hart. *The Alpha–beta Heuristic (AIM-030).* 1961. Massachusetts Institute of Technology.

[23] T. Cormen, et al. *Introduction to Algorithms.* 2009. MIT Press.

[24] I. Roizen, P. Judea. *A minimax algorithm better than alpha-beta?: Yes and No.* 1983. Artificial Intelligence.

[25] D. Delling, P. Sanders, D. Schultes, D. Wagner. *Engineering Route Planning Algorithms.* 2009. Algorithmics of Large and Complex Networks: Design, Analysis, and Simulation. Lecture Notes in Computer Science. Springer.

[26] S. LaValle. *Rapidly-exploring random trees: A new tool for path planning.* 1998. Technical Report.

Computer Science Department, Iowa State University.

[27] R. Korf. *Depth-first Iterative-Deepening: An Optimal Admissible Tree Search.* 1985. Artificial Intelligence.

[28] T. Lai, H. Robbins. *Asymptotically efficient adaptive allocation rules.* 1985. Advances in Applied Mathematics.

[29] P. Auer, N. Cesa-Bianchi, P. Fisher. *Finite-time analysis of the multiarmed bandit problem.* 2002. Machine Learning.

[30] B. Abramson *The Expected-Outcome Model of Two-Player Games.* 1987. Technical report, Department of Computer Science, Columbia University.

[31] B. Brügmann. *Monte Carlo Go.* 1993. Technical report, Department of Physics, Syracuse University.

[32] W. Ertel, J. Schumann, C. Suttner. *Learning Heuristics for a Theorem Prover using Back Propagation.* 1989. Österreichische Artificial-Intelligence-Tagung. Informatik-Fachberichte. Springer.

[33] D. Silver. *Mastering the game of Go with deep neural networks and tree search.* 2016. Nature.

[34] H. Chang; M. Fu, J. Hu, S. Marcus. *An Adaptive Sampling Algorithm for Solving Markov Decision Processes.* 2005. Operations Research.

[35] C. Rémi. *The Monte-Carlo Revolution in Go.* 2008. Japanese-French Frontiers of Science Symposium.

[36] L. Kocsis and C. Szepesvári *Bandit based Monte-Carlo Planning.* 2006. Machine Learning: ECML 2006, 17th European Conference on Machine Learning. Proceedings. Lecture Notes in Computer Science.

[37] C. Rémi. *Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search.* 2007. Computers and Games, 5th International Conference, CG 2006.

[38] P. Auer, N. Cesa-Bianchi, P. Fischer. *Finite-time Analysis of the Multiarmed Bandit Problem.* 2002. Machine Learning.

[39] H. Chang, M. Fu, Jiaqiao Hu, S. Marcus. *Google DeepMind's Alphago: O.R.'s unheralded role in the path-breaking achievement.* 2016. OR/MS Today. 45 (5): 24–29.

[40] Y. Peres, O. Schramm, S. Sheffield, D. Wilson. *Random-Turn Hex and other selection games.* 2006. arXiv:math/0508580.

[41] D. Shah, Q. Xie, Z. Xu. *Non-Asymptotic Analysis of Monte Carlo Tree Search.* 2020. arXiv:1902.05213v.

[42] T. Vodopivec, S. Samothrakis, B. Ster. *On Monte Carlo Tree Search and Reinforcement Learning.* 2017. Journal of Artificial Intelligence Research 60.

[43] B. Kartal, J. Koenig, S. J. Guy. *User-driven narrative variation in large story domains using monte carlo tree search.* 2014. In Proceedings of the 2014 International Conference on Autonomous Agents and Multi-agent Systems.

Monte Carlo Tree Search for Tic-Tac-Toe

Peter Racioppo

In [1]:
```
# Imports

import numpy as np
import matplotlib.pyplot as plt
from pylab import rcParams

import random
import math
from scipy.linalg import eig
```

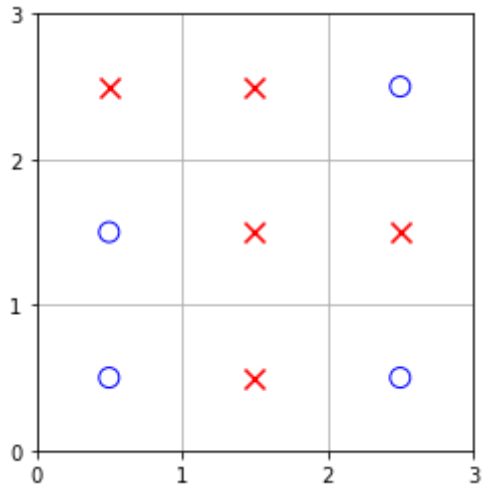
Basic board functionality and a function to play random games:

In [2]:
```
# This function plots the tic-tac-toe board:
def f_Plot_board(M, Winner):
    # Inputs:
    # M, a matrix which keeps track of pieces on the board
    # Winner, an indicator for who one (1 for X, -1 for O, 0 for draw)

    # Print who won:
    if Winner == 1:
        print("X wins!")
    elif Winner == -1:
        print("O wins!")
    else:
        print("Draw!")

    fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1)
    plt.gca().set_aspect('equal', adjustable='box')

    # Loop through the boxes in the grid and plot the pieces on the board:
    # Note: Our matrix is indexed from left to right and from top to bottom,
    # but our board is indexed from left to right and from bottom to top.
    n = np.shape(M)[0]
    for i in np.arange(n):
        for j in np.arange(n):
            if M.T[i,n-1-j] == 1:
                plt.scatter(i+0.5,j+0.5,s=100,c="r",marker="x")
            elif M.T[i,n-1-j] == -1:
                plt.scatter(i+0.5,j+0.5,s=100,marker="o",facecolors='none', edgecolors='b')

    plt.xlim(0, n)
    plt.ylim(0, n)
    major_ticks = np.arange(0, n+1, 1)
    ax.set_xticks(major_ticks)
    ax.set_yticks(major_ticks)
    ax.grid(which='both')
    plt.show()
```

In [4]:
```
# Plays a random game of Tic-tac-toe on an n x n board:
def f_Rand_TTT(n=3):
    # Input: n, the board dimension.

    pos = list(range(n**2)) # List of possible moves

    # v is a list version of the matrix representing the board.
    # It holds the moves that have been made in each position:
    # 1 for X, -1 for O, 0 for no piece.
    v = np.zeros(n**2)
    # While there is a valid move:
    for i in np.arange(n**2):
        move = np.random.choice(pos) # Choose a random move from the remaining possibilities
        idx = pos.index(move)
        del pos[idx] # Remove our new move from the list of possibilities
        # Append the move to the list of moves:
        if np.mod(i,2) == 0:
            v[move] = 1
        else:
            v[move] = -1

        # Check if someone won:
        M = np.reshape(v,[n,n]) # Reshape the list to a matrix
        Winner = 0 # Default to draw
        # If X occupies an entire row or an entire column or the main diagonal or the off diagonal, X w
        if n in M.sum(axis=1) or n in M.sum(axis=0) or np.trace(M) == n or np.trace(np.fliplr(M)) == n:
            Winner = 1
            break
        # If O occupies an entire row or an entire column or the main diagonal or the off diagonal, O w
        elif -n in M.sum(axis=1) or -n in M.sum(axis=0) or np.trace(M) == -n or np.trace(np.fliplr(M)) =
            Winner = -1
            break

    # Outputs:
    # M, a matrix which keeps track of pieces on the board
    # Winner, an indicator for who one (1 for X, -1 for O, 0 for draw)
    return M, Winner
```

In [5]:
```
# Play a Random Game:
n = 3
M, Winner = f_Rand_TTT(n)
f_Plot_board(M,Winner)

print(M)
```

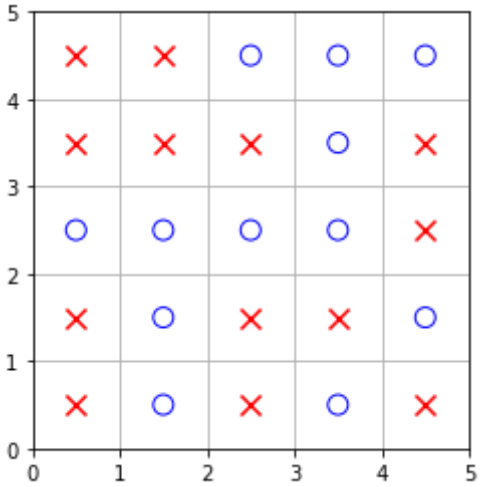
```
X wins!
```




```
[[ 1.  1. -1.]
 [-1.  1.  1.]
 [-1.  1. -1.]]
```

In [6]:
```
# Our functions support arbitrary board size:
n = 5
M, Winner = f_Rand_TTT(n)
f_Plot_board(M,Winner)
```

```
Draw!
```



In [3]:
```
# Play N random games and calculate the percent won/lost/tied
def f_Play_N_Random_Games(N=1000,n=3):

    Win_Count = [] # Number of wins/losses/draws
    for i in np.arange(N):
        _, Winner = f_Rand_TTT(n) # Play a game and record Winner
        Win_Count.append(Winner) # Append Winner to list

    X_wins = np.shape(np.array(Win_Count)[np.where(np.array(Win_Count) == 1)])[0] # Total number of win
    O_wins = np.shape(np.array(Win_Count)[np.where(np.array(Win_Count) == -1)])[0] # Total number of wi
    draws = np.shape(np.array(Win_Count)[np.where(np.array(Win_Count) == 0)])[0] # Total number of draw

    percent_X_wins = 100*X_wins/N
    percent_O_wins = 100*O_wins/N
    percent_draws = 100*draws/N

    print("Percent X wins =", percent_X_wins, "%")
    print("Percent O wins =", percent_O_wins, "%")
    print("Percent draws =", percent_draws, "%")
```

In [8]:
```
f_Play_N_Random_Games(N=10000,n=3)
```

```
Percent X wins = 58.34 %
Percent O wins = 28.85 %
Percent draws = 12.81 %
```

Player X (who goes first) wins about 60% of games. Player O (who goes second) wins about 30% of games. This will be our baseline performance.

Now we build a simple AI:

Let's begin with a simple Monte Carlo search that considers only the next move and then plays random games starting from each possible next move. The random games are played with replacement. For each move, we compute the total number of wins minus the total number of losses in our random games. We then pick the move that maximizes the number of wins minus the number of losses. This process amounts to maximizing an empirical estimate of the expectation of winning.

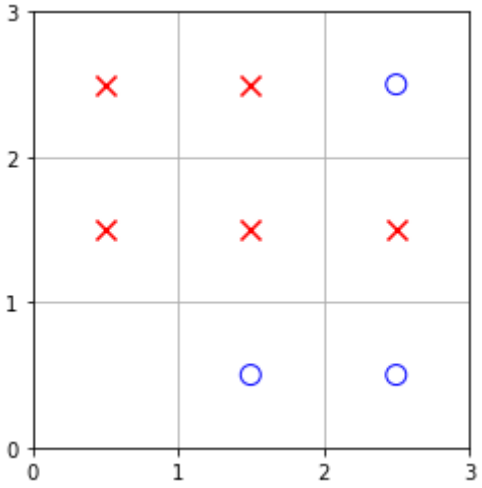
In [4]:
```
# Plays a random game starting from a position
# specified by prev_x, prev_o, the positions where
# x's and o's have already been played.
def f_Rand_i(n,x_moves=[],o_moves=[]):

    pos_i = list(range(n**2)) # List of all possible moves
    prev_moves = x_moves + o_moves # List of all previous moves
    pos_i = list(np.delete(pos_i,prev_moves)) # Remove previous moves from possible moves

    v = np.zeros(n**2)
    v[x_moves] = 1
    v[o_moves] = -1

    M_i = np.reshape(v,[n,n]) # Reshape the list to a matrix

    Winner_i = 0
    for i in np.arange(np.shape(pos_i)[0]):
        move = np.random.choice(pos_i) # Choose a random move from the possibilities
        idx = pos_i.index(move)
        del pos_i[idx]
        if np.mod(i,2) == 0:
            v[move] = 1
            x_moves.append(move)
        else:
            v[move] = -1
            o_moves.append(move)

        # Check if someone won:
        M_i = np.reshape(v,[n,n]) # Reshape the list to a matrix
        if n in M_i.sum(axis=1) or n in M_i.sum(axis=0) or np.trace(M_i) == n or np.trace(np.fliplr(M_i
            Winner_i = 1
            break
        elif -n in M_i.sum(axis=1) or -n in M_i.sum(axis=0) or np.trace(M_i) == -n or np.trace(np.flipl
            Winner_i = -1
            break

    return x_moves, o_moves, M_i, Winner_i
```

In [14]:
```
# Test

n = 3
pos = list(range(n**2))
prev_moves = [0,1,2]
pos = list(np.delete(pos,prev_moves))
pos
```

Out[14]:
```
[3, 4, 5, 6, 7, 8]
```

In [15]:
```
# Test

n = 3
x_moves = [0,1]
o_moves = [2]

x_moves, o_moves, M, Winner = f_Rand_i(n,x_moves,o_moves)
f_Plot_board(M,Winner)
```

```
X wins!
```



In [5]:
```
# Appends a test_move to the list of moves that have already been made.
def f_Append_Test_Move(x_moves,o_moves,test_move,n):

    # Create copies to append the test value to:
    x_moves_test = list(np.empty_like(x_moves))
    o_moves_test = list(np.empty_like(o_moves))
    x_moves_test[:] = x_moves
    o_moves_test[:] = o_moves

    # If it's x's turn, append the test move to the list of x_moves
    # (x went first)
    if len(x_moves) <= len(o_moves):
        x_moves_test += [test_move]
    # If it's o's turn, append the test move to the list of o_moves
    else:
        o_moves_test += [test_move]

    return x_moves_test, o_moves_test
```

In [16]:
```
# Test

x_moves = [0,1]
o_moves = [2]

test_move = [np.random.choice(pos)] # Choose a test move at random
x_moves_test, o_moves_test = f_Append_Test_Move(x_moves,o_moves,test_move,n)
print(x_moves_test)
print(o_moves_test)
```

```
[0, 1]
[2, [4]]
```

In [6]:
```
# Branch, with replacement, from test move
def f_Branch_from_test_move(x_moves,o_moves,test_move,n=3,it=100):
    x_moves_test, o_moves_test = f_Append_Test_Move(x_moves,o_moves,test_move,n)
    count = 0
    for i in np.arange(it):
        x_moves_i = list(np.empty_like(x_moves_test))
        o_moves_i = list(np.empty_like(o_moves_test))
        x_moves_i[:] = x_moves_test
        o_moves_i[:] = o_moves_test
        _, _, _, Winner_i = f_Rand_i(n,x_moves_i,o_moves_i)
        count += Winner_i

    return count
```

In [7]:
```
# Given a list of possible moves, we consider
# all posibilites for the next move. We then
# randomly branch on each of these moves,
# and return the number of wins minus the
# number of losses for each.
def f_Monte_Carlo_Outcomes(pos,x_moves,o_moves,n,it=100):
    pos_test = list(np.empty_like(pos))
    pos_test[:] = pos
    L = np.shape(pos_test)[0]
    Count_v = np.zeros(L)

    for l in np.arange(L):
        test_move = pos_test[l]
        count = f_Branch_from_test_move(x_moves,o_moves,test_move,n=n,it=it)
        Count_v[l] = count

    return Count_v
```

In [55]:
```
Count_v = f_Monte_Carlo_Outcomes(pos,x_moves,o_moves,n)
print(Count_v)
```

```
[ 41. -31.  14.  -8.  33.  -9.]
```

In [21]:
```
# Test
# In what percent of games does X choose the center position as the first move?

n = 3
it = 100
O_mode = 0
X_mode = 1

counter = 0
iterations = 100

pos = list(range(n**2)) # List of all possible moves

for i in np.arange(iterations):

    x_moves = []
    o_moves = []

    if (len(x_moves) <= len(o_moves)) and X_mode == 1:
        Count_v = f_Monte_Carlo_Outcomes(pos,x_moves,o_moves,n,it)
        idx = np.argmax(Count_v)
    elif (len(x_moves) > len(o_moves)) and O_mode == 1:
        Count_v = f_Monte_Carlo_Outcomes(pos,x_moves,o_moves,n,it)
        idx = np.argmin(Count_v)
    else:
        idx = np.random.choice(np.arange(np.shape(pos)[0]))

    new_move = pos[idx]

    if new_move == 4:
        counter += 1

print(counter/iterations)
```

```
0.89
```

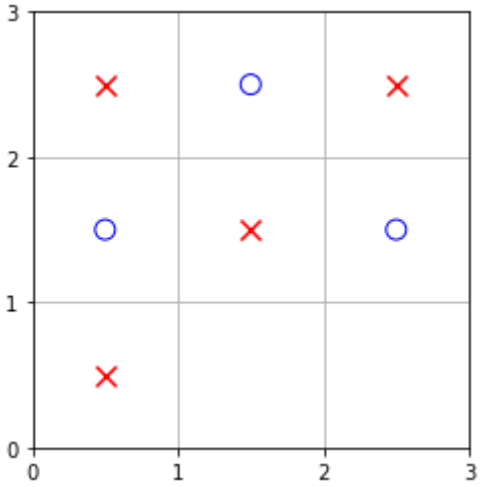
Player 1 chooses the center box as its first move about 90% of the time.

In [8]:
```
# Plays a game of Tic-tac-toe on an n x n board,
# using a pure Monte Carlo search algorithm.
def f_Monte_Carlo_TTT(X_mode,O_mode,n=3,it=100):
    # Inputs:
    # X_mode = 1 for Monte Carlo algorithm, X_mode = 0 for random play
    # O_mode = 1 for Monte Carlo algorithm, O_mode = 0 for random play
    # n, the board dimension.
    # it, the number of MC simulations for each test move

    pos = list(range(n**2)) # List of possible moves

    # v is a list version of the matrix representing the board.
    # It holds the moves that have been made in each position:
    # 1 for X, -1 for O, 0 for no piece.
    v = np.zeros(n**2)

    x_moves = []
    o_moves = []

    Winner = 0 # Default to draw

    # While there is a valid move:
    for i in np.arange(n**2):
        if np.mod(i,2) == 0 and X_mode == 1:
            Count_v = f_Monte_Carlo_Outcomes(pos,x_moves,o_moves,n,it)
            idx = np.argmax(Count_v)
        elif np.mod(i,2) == 1 and O_mode == 1:
            Count_v = f_Monte_Carlo_Outcomes(pos,x_moves,o_moves,n,it)
            idx = np.argmin(Count_v)
        else:
            idx = np.random.choice(np.arange(np.shape(pos)[0]))

        new_move = pos[idx]
        del pos[idx]

        # Append the move to the list of moves:
        if np.mod(i,2) == 0:
            x_moves += [new_move]
            v[new_move] = 1
        else:
            o_moves += [new_move]
            v[new_move] = -1
```

```
        # Check if someone won:
        M = np.reshape(v,[n,n]) # Reshape the list to a matrix
        # If X occupies an entire row or an entire column or the main diagonal or the off diagonal, X w
        if n in M.sum(axis=1) or n in M.sum(axis=0) or np.trace(M) == n or np.trace(np.fliplr(M)) == n:
            Winner = 1
            break
        # If O occupies an entire row or an entire column or the main diagonal or the off diagonal, O w
        elif -n in M.sum(axis=1) or -n in M.sum(axis=0) or np.trace(M) == -n or np.trace(np.fliplr(M)) =
            Winner = -1
            break

    # Outputs:
    # M, a matrix which keeps track of pieces on the board
    # Winner, an indicator for who won (1 for X, -1 for O, 0 for draw)
    return M, Winner
```

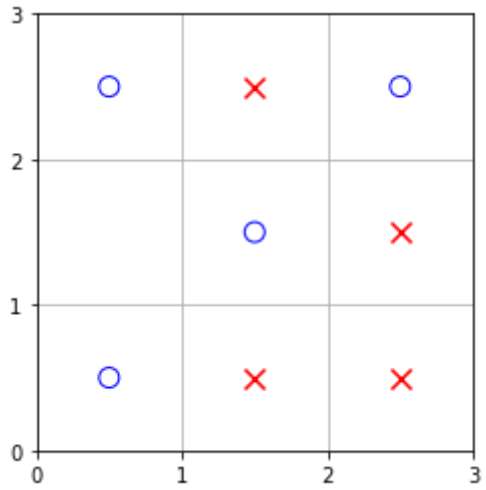
In [9]:
```
M, Winner = f_Monte_Carlo_TTT(X_mode=1,O_mode=0,n=3,it=100)
f_Plot_board(M,Winner)
```

```
X wins!
```



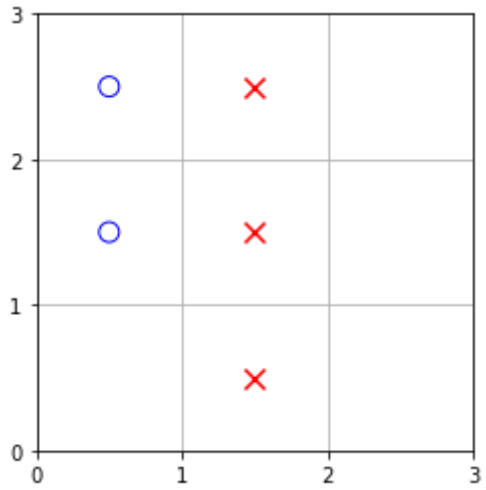
In [10]:
```
M, Winner = f_Monte_Carlo_TTT(X_mode=0,O_mode=1,n=3,it=10)
f_Plot_board(M,Winner)
```

```
O wins!
```



In [12]:
```
M, Winner = f_Monte_Carlo_TTT(X_mode=1,O_mode=1,n=3,it=10)
f_Plot_board(M,Winner)
```

```
X wins!
```



In [9]:
```
# Play N games and calculate the percent won/lost/tied
# (using a random Monte Carlo search)
def f_Play_N_Games(X_mode,O_mode,n=3,it=100,N=100):

    Win_Count = [] # Number of wins/losses/draws
    for i in np.arange(N):
        _, Winner = f_Monte_Carlo_TTT(X_mode=X_mode,O_mode=O_mode,n=n,it=it) # Play a game and record W
        Win_Count.append(Winner) # Append Winner to list

    X_wins = np.shape(np.array(Win_Count)[np.where(np.array(Win_Count) == 1)])[0] # Total number of win
    O_wins = np.shape(np.array(Win_Count)[np.where(np.array(Win_Count) == -1)])[0] # Total number of wi
    draws = np.shape(np.array(Win_Count)[np.where(np.array(Win_Count) == 0)])[0] # Total number of draw

    percent_X_wins = 100*X_wins/N
    percent_O_wins = 100*O_wins/N
    percent_draws = 100*draws/N

    print("Percent X wins =", percent_X_wins, "%")
    print("Percent O wins =", percent_O_wins, "%")
    print("Percent draws =", percent_draws, "%")
```

We now test the algorithm by simulating tournaments.

10,000 games with both players playing randomly:

In [801]:
```
f_Play_N_Games(X_mode=0,O_mode=0,n=3,it=100,N=10000)
```

```
Percent X wins = 58.15 %
Percent O wins = 29.04 %
Percent draws = 12.81 %
```

100 games with X using the Monte Carlo algorithm and O playing randomly:

In [802]:
```
f_Play_N_Games(X_mode=1,O_mode=0,n=3,it=100,N=100)
```

```
Percent X wins = 91.0 %
Percent O wins = 7.0 %
Percent draws = 2.0 %
```

In [804]:
```
f_Play_N_Games(X_mode=1,O_mode=0,n=3,it=10,N=100)
```

```
Percent X wins = 88.0 %
Percent O wins = 12.0 %
Percent draws = 0.0 %
```

Decreasing the number of Markov chains per simulation from 100 to 10 seems to only marginally worsen performance.

100 games with O using the Monte Carlo algorithm and X playing randomly:

In [13]:
```
f_Play_N_Games(X_mode=0,O_mode=1,n=3,it=100,N=100)
```

```
Percent X wins = 2.0 %
Percent O wins = 74.0 %
Percent draws = 24.0 %
```

When 10 Markov chains are used, O wins about 70% of the time.

In [809]:
```
f_Play_N_Games(X_mode=0,O_mode=1,n=3,it=10,N=100)
```

```
Percent X wins = 12.0 %
Percent O wins = 71.0 %
Percent draws = 17.0 %
```

---

100 games with both players using the Monte Carlo algorithm:

In [787]:
```
f_Play_N_Games(X_mode=1,O_mode=1,n=3,it=10,N=100)
```

```
Percent X wins = 49.0 %
Percent O wins = 40.0 %
Percent draws = 11.0 %
```

With both players using the Monte Carlo algorithm, X still wins more, due to its starting advantage, but its advantage is much smaller than in the random case.

To improve the algorithm, we can progressively narrow the window of the Monte Carlo search to spend extra time searching on the most promising moves, first searching over all n moves, then the

best n/k moves, then the best n/k^2 moves, and so on.

In [178]:
```
# Plays a game of Tic-tac-toe on an n x n board,
# using a pure Monte Carlo search algorithm.
# Allows for the window for the Monte Carlo search
# to be progressively decreased in size.
def f_Exp_Window_Monte_Carlo_TTT(X_mode,O_mode,n=3,window=1,it=100):
    # Inputs:
    # X_mode = 1 for Monte Carlo algorithm, X_mode = 0 for random play
    # O_mode = 1 for Monte Carlo algorithm, O_mode = 0 for random play
    # n, the board dimension.
    # window, the number of windows to use in search. window = 1 for 1 MC search,
    # window = 2 for a repeated search on the 4 best moves, window = 3 for a third
    # search on the best 2 moves.
    # it, the number of MC simulations for each test move

    pos = list(range(n**2)) # List of possible moves

    # v is a list version of the matrix representing the board.
    # It holds the moves that have been made in each position:
    # 1 for X, -1 for O, 0 for no piece.
    v = np.zeros(n**2)

    x_moves = []
    o_moves = []

    Winner = 0 # Default to draw

    # While there is a valid move:
    for i in np.arange(n**2):
        if np.mod(i,2) == 0 and X_mode == 1:
            Count_v = f_Monte_Carlo_Outcomes(pos,x_moves,o_moves,n,it)
            # If there are at least 4 possible moves remaining,
            # repeat the pure Monte Carlo search on the 4 best moves.
            if np.shape(Count_v)[0] > 4 and window >= 2:
                idxs = np.flip(np.argsort(np.array(Count_v)))[0:4]
                pos_i = np.array(pos)[idxs]
                Count_v = f_Monte_Carlo_Outcomes(pos_i,x_moves,o_moves,n,it)
                # If there are at least 2 possible moves remaining,
                # repeat the pure Monte Carlo search on the 2 best moves.
                if np.shape(Count_v)[0] > 2 and window >= 3:
                    idxs2 = np.flip(np.argsort(np.array(Count_v)))[0:2]
                    pos_i = np.array(pos)[idxs2]
                    Count_v = f_Monte_Carlo_Outcomes(pos_i,x_moves,o_moves,n,it)
```

```
                idxk = np.argmax(Count_v)
                idxj = idxs2[idxk]
            else:
                idxj = np.argmax(Count_v)
            idx = idxs[idxj]
        else:
            idx = np.argmax(Count_v)
    elif np.mod(i,2) == 1 and O_mode == 1:
        Count_v = f_Monte_Carlo_Outcomes(pos,x_moves,o_moves,n,it)
        # If there are at least 4 possible moves remaining,
        # repeat the pure Monte Carlo search on the 4 best moves.
        if np.shape(Count_v)[0] >= 4 and window >= 2:
            idxs = np.argsort(np.array(Count_v))[0:4]
            pos_i = np.array(pos)[idxs]
            Count_v = f_Monte_Carlo_Outcomes(pos_i,x_moves,o_moves,n,it)
            # If there are at least 2 possible moves remaining,
            # repeat the pure Monte Carlo search on the 2 best moves.
            if np.shape(Count_v)[0] >= 2 and window >= 3:
                idxs2 = np.argsort(np.array(Count_v))[0:2]
                pos_i = np.array(pos)[idxs2]
                Count_v = f_Monte_Carlo_Outcomes(pos_i,x_moves,o_moves,n,it)
                idxk = np.argmin(Count_v)
                idxj = idxs2[idxk]
            else:
                idxj = np.argmin(Count_v)
            idx = idxs[idxj]
        else:
            idx = np.argmin(Count_v)
    else:
        idx = np.random.choice(np.arange(np.shape(pos)[0]))

    new_move = pos[idx]
    del pos[idx]

    # Append the move to the list of moves:
    if np.mod(i,2) == 0:
        x_moves += [new_move]
        v[new_move] = 1
    else:
        o_moves += [new_move]
        v[new_move] = -1

    # Check if someone won:
```

```
        M = np.reshape(v,[n,n]) # Reshape the list to a matrix
        # If X occupies an entire row or an entire column or the main diagonal or the off diagonal, X w
        if n in M.sum(axis=1) or n in M.sum(axis=0) or np.trace(M) == n or np.trace(np.fliplr(M)) == n:
            Winner = 1
            break
        # If O occupies an entire row or an entire column or the main diagonal or the off diagonal, O w
        elif -n in M.sum(axis=1) or -n in M.sum(axis=0) or np.trace(M) == -n or np.trace(np.fliplr(M)) =
            Winner = -1
            break

    # Outputs:
    # M, a matrix which keeps track of pieces on the board
    # Winner, an indicator for who won (1 for X, -1 for O, 0 for draw)
    return M, Winner
```

In [150]:
```
# Compute the percentage of games in which X plays the center position as its first move:

iterz = 100
count = 0
for i in np.arange(iterz):
    M, Winner = f_Exp_Window_Monte_Carlo_TTT(X_mode=1,O_mode=1,n=3,it=10)

    if M[1,1] == 1:
        count += M[1,1]
count/iterz
```

Out[150]: 57.0

In [165]:
```
 N random games and calculate the percent won/lost/tied
ing the windowed-version of the Monte Carlo search)
_Play_N_Games2(X_mode,O_mode,n=3,it=100,window=1,N=100):

n_Count = [] # Number of wins/losses/draws
or i in np.arange(N):
   _, Winner = f_Exp_Window_Monte_Carlo_TTT(X_mode=X_mode,O_mode=O_mode,n=n,window=window,it=it) # Play a
   Win_Count.append(Winner) # Append Winner to list

_wins = np.shape(np.array(Win_Count)[np.where(np.array(Win_Count) == 1)])[0] # Total number of wins for X
_wins = np.shape(np.array(Win_Count)[np.where(np.array(Win_Count) == -1)])[0] # Total number of wins for
raws = np.shape(np.array(Win_Count)[np.where(np.array(Win_Count) == 0)])[0] # Total number of draws

ercent_X_wins = 100*X_wins/N
ercent_O_wins = 100*O_wins/N
ercent_draws = 100*draws/N

rint("Percent X wins =", percent_X_wins, "%")
rint("Percent O wins =", percent_O_wins, "%")
rint("Percent draws =", percent_draws, "%")
```

In [176]:
```
f_Play_N_Games2(X_mode=1,O_mode=0,n=3,it=100,window=1,N=100)
```

```
Percent X wins = 95.0 %
Percent O wins = 5.0 %
Percent draws = 0.0 %
```

In [158]:
```
f_Play_N_Games2(X_mode=1,O_mode=0,n=3,it=100,window=2,N=100)
```

```
Percent X wins = 95.0 %
Percent O wins = 4.0 %
Percent draws = 1.0 %
```

In [153]:
```
f_Play_N_Games2(X_mode=1,O_mode=0,n=3,it=100,window=3,N=100)
```

```
Percent X wins = 90.0 %
Percent O wins = 10.0 %
Percent draws = 0.0 %
```

In [177]:
```
f_Play_N_Games2(X_mode=0,O_mode=1,n=3,it=100,window=1,N=100)
```

```
Percent X wins = 1.0 %
Percent O wins = 76.0 %
Percent draws = 23.0 %
```

In [160]:
```
f_Play_N_Games2(X_mode=0,O_mode=1,n=3,it=100,window=2,N=100)
```

```
Percent X wins = 3.0 %
Percent O wins = 65.0 %
Percent draws = 32.0 %
```

In [154]:
```
f_Play_N_Games2(X_mode=0,O_mode=1,n=3,it=100,window=3,N=100)
```

```
Percent X wins = 18.0 %
Percent O wins = 63.0 %
Percent draws = 19.0 %
```

In [ ]:
```
# The window method actually seems to make things worse.
```