Applying Deep Reinforcement Learning to Finite State Single Player Games

Donald Stephens
Department of Computer Science
Stanford University
Stanford, CA 94305
dsteph@stanford.edu

Abstract

Reinforcement Learning is an area of Machine Learning where a learner commonly known as an agent interacts with its environment. The agent is not told what actions to take, rather it must discover them through exploration. The agent is lead by the goal of yielding rewards and is satisfied when it has maximized its reward. We developed a Deep Reinforcement Learning model based on Deep Q-Learning (DQN) to teach an agent to solve any puzzle of Solitaire Chess.

1 Background

1.1 Game Summary

Solitaire Chess is a single-player logic game utilizing the same rules of classical Chess but presented in a simplified form posed as mini Chess problems rather than full length opponent-based strategy games. A chess problem is a puzzle to be solved using a chess board (of a specified dimension) and a subset of standard Chess pieces. We outlined the rules of Solitaire Chess in the Appendix section of this document.

2 Task Definition

2.1 Our Task

Design an engine capable of solving a set of Chess-based logic games known as Solitaire Chess. In particular, the engine should not only solve any validly constructed puzzle; but, also be capable of performing better than a brute force approach. A brute force approach attempts to solve a puzzle by enumerating every possible move of the chess pieces until a solution is found. For our engine to be considered better, the engine should arrive at the correct target solution with less exploration of possible moves. We established performance metrics to evaluate our engine compared to solutions constructed by the author of Solitaire Chess, a human model, which we used as our oracle model.

2.2 Scope

The engine was given a board of a specified square dimension, and a subset of chess pieces with their respectful locations on the board (collectively known as *inputs*). We used a fixed sized 4 by 4 board which coincided with the parameters of the human generated dataset of solutions. The engine should solve any validly constructed puzzle by identifying the final remaining board piece along with a detailed list of steps, which will be a sequence of chess moves. The winning chess piece along with the chess moves will collectively be known as *outputs*. The given dataset contained a unique final remaining board piece for each puzzle. However, in some cases, there were multiple paths that led to

the unique ultimate outcome. To assure consistent evaluation and experimentation, our evaluation metrics excluded comparisons of paths as part of the performance criterion.

2.3 Evaluation Metrics

To avoid subjective or arbitrary considerations on performance, we defined two key metrics to evaluate our engine:

1. The **accuracy** of identifying the intended final board piece and its final position. At minimum, our engine should achieve an accuracy of 100% to be considered a viable approach.

$$accuracy = \frac{\#\ number\ of\ matching\ valid\ solution}{\#\ number\ of\ puzzles}$$

2. The **average number of paths explored** before identifying a valid solution. Though we use the term "paths", a more appropriate term here would be "steps" or "moves".

The total amount of (wall clock) time needed to solve a puzzle will not be considered as part of our performance criterion. To be irrespective of machine processing power and parallel computation (if any), we believe that the number of paths explored is a reasonable measure of model runtime performance.

2.4 Dataset

We acquired a dataset of eighty puzzles (categorized by four levels of difficulty) each containing:

- the initial board setup
- identification of the final board piece
- and a solution defined by a set of moves

The pre-set puzzles were made available by the game's author on the ThinkFun website (`http://www.thinkfun.com`).

3 Approach

3.1 Modeling

The modeling approaches explored for our project were all based on the notion of state-based modeling. To initiate our discussion, it's prudent to introduce the following terminology:

- Agent: An entity that determines what decisions to make. Our project assumed the presence of a single agent where our model acts as the agent.
- Environment: A collection of the following: a 4 x 4 chess board, a subset of predefined chess pieces, and a response to a decision made by the agent. The model of the environment was governed by the rules of Solitaire Chess.
- Time-step: An incremental change in time characterized by a move of a chess piece. The original board setup at the start of the game will be denoted as t_0. Tracking of steps through time can be facilitated as an offset from t_0 such as $t_0 + 3$ (which denotes 3 moves after game start) or simply 3 if the given context is clear.
- State: The condition of the environment at a particular time-step. The state will be composed of a 4 x 4 chess board with a set of chess pieces positioned on the board. Generally, the set of states will be denoted as $States$. Hence, a current state may be denoted as $s \in States$. As such, a state at a particular time-step, t, may be denoted as s_t.
- Action: The set of all valid moves for each chess piece in a given state. A valid action is one that reduces the number of chess pieces in the current state by 1 should the action be taken. Generally, the set of actions from a given state, $s \in States$, will be denoted as $Actions_s$. Hence, a specific action may be denoted as $a \in Actions_s$.
- Reward: Identifies the immediate intrinsic value of a desired state.

- Value: Identifies the (long term) total amount of reward an agent can expect to accumulate in the future stemming from the current state. Generally, the value stemming from the current state,$s \in States$ will be denoted as $V(s)$.
- Policy: A mapping of perceived states of the environment to actions to be taken when in those states. It defines the behavior the agent should follow based on the current state. Generally, a policy will be denoted as π. If π and $\pi^{'}$ are two distinct policies, we define π to be better than or equal to $\pi^{'}$ if an only if $V_{\pi}(s) >= V_{\pi^{'}}(s)\ \forall s \in States$. An optimal policy will be denoted as π^{*} and will be one such that $V_{\pi^{*}}(s) = \max_{\pi}\{V_{\pi}(s)\}\ \forall s \in States$.

In our modeling experiments, we explored approaches based on dynamic programming as well as some based on neural networks.

3.2 Baseline (Benchmark) Model

As a baseline, we have implemented a solver that iterates through all combinations of valid moves until an appropriate solution is identified. The technique employs no heuristics neither any other optimization technique. It always provides the correct answer but will tend not to be efficient on average.

This model is implemented as a recursive routine that utilizes the following steps:

1. For every state of the board, the solver identifies the pieces on the board by traversing from the bottom row to the top row, and from the right column to the left column for each row. When it encounters a piece, it stores that piece within a vector.
2. For each piece in the pieces vector, the solver will identify all of the valid moves that the piece can make from that position. Valid moves are determined by the intersection of rules from Classical Chess and Solitaire Chess. Please note that every move must result in a piece capture.
3. Using the vector of moves identified in Step 2, the solver selects a move by iterating backwards in that vector. It uses the selected move to capture a piece and transition to Step 1 to begin a new cycle, continuing onwards until no more moves are available or a valid solution is found.

Since the baseline is not intended to be optimized, the choice of selecting moves based on their reversed ordinal positions is arbitrary. The baseline model is akin to a traditional backtracking algorithmic approach.

3.3 Oracle Model

As an oracle, we chose to use a human model. Specifically, the author of Solitaire Chess composed a number of solutions by hand, categorized by four levels of difficulty. Each handcrafted solution contained the initial board setup, identification of the final board piece, and a solution defined by a set of moves. Note that all of the puzzle solutions in our dataset were provided by the oracle model.

3.4 Proposed Model

3.4.1 Background

Each puzzle requires moving chess pieces sequentially, capturing a piece on each move, until a single chess piece remained. Following our terminology provided in the modeling section above, we modeled the game play as a finite state single player problem. Prior to a move, one or more chess pieces could take an action to capture another piece. The goal is to reach the end state (a single remaining chess piece) by taking the appropriate actions at each state. Our proposed model is based on deep reinforcement learning (DRL) which is based on Deep Q-Learning (DQN) to calculate the Q-values of the actions.

In passing we note that even though Solitaire Chess is a variant of Classical Chess, we admit that popular artificial Chess-solving techniques such as variants of naive minimax, and alpha-beta pruning are not applicable here. Such techniques consider future states (at least 2 steps into the future) when determining optimal actions to take in a given state. For Solitaire Chess:

- there are no opposing agents
- all moves must result in a captured piece to be considered a valid move

3.4.2 Description

The benefit of the baseline model is that it always gives the correct (as in expected) set of results when given valid input data. A correct result is defined by the correct identification of the intended final remaining chess piece. However, the drawback is in the number of potential moves the model may explore before arriving at a valid solution. This limitation is based on the fact that the baseline model makes no attempt to reduce the search space nor does it attempt to rank a given set of actions in an optimal way.

Before diving into the details of the DRL model, let us first review an example taken from the baseline model's results. Consider Figure 1, which is a visual depiction of the solution steps for Puzzle 2 in our Oracle generated solutions dataset. We numerically labeled each step taken by the model in order. To solve this puzzle, the baseline model required 11 steps.

As described in the Baseline (Benchmark) Model section, the first step the model does is to order the pieces on the board. However, the order taken is not done by any design nor strategy. The order was relative to the position of the pieces on the board. In this example, the model ordered the pieces as follows: PAWN, BISHOP, KNIGHT, and ROOK. This is also evident by the blue lines in the second level of the tree. It is important to note here that had the model ordered ROOK higher in the initial ordering, a lower number of failed steps would've occurred.

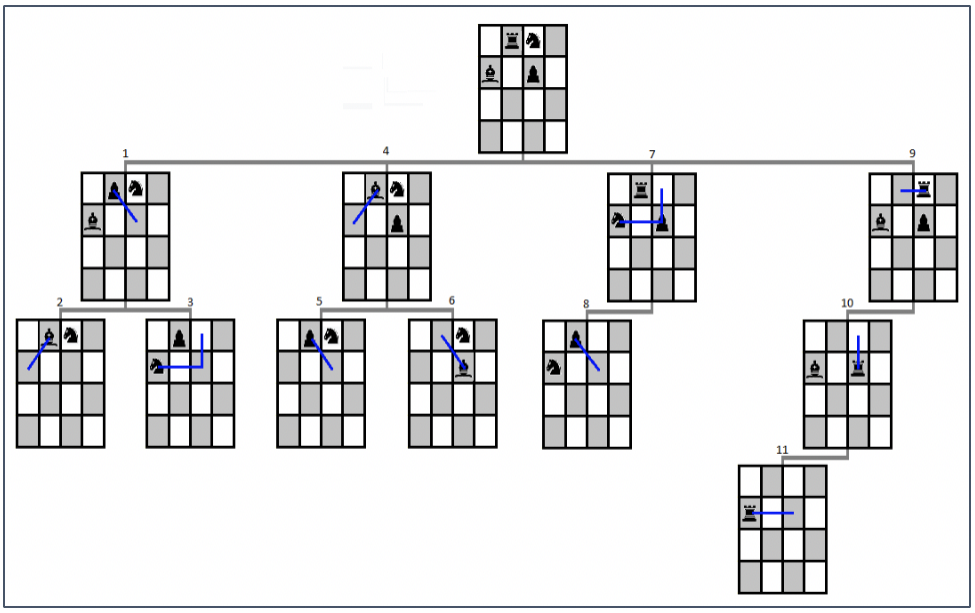

Figure 1: Backtracking for Puzzle #2

3.4.3 Deep Reinforcement Learning (DRL)

Deep reinforcement learning entails using an artificial neural network with 1 or more hidden layers to learn which actions to take in given states through an exploratory trial-and-error process. The two benefits of such an approach are summarized as:

- Trial-and-error search: a search process that allows us to explore options that may not be as rewarding at the moment but possibly more rewarding in the long run.
- Delayed reward: assessment of subsequent rewards

Our was based on Deep Q-Learning (DQN); a technique aimed at choosing the best action for a given set of circumstances. In our case, the circumstance is the relative position of the pieces on the board

(which is encoded in the state). Each possible action that could be taken from a given state is given an estimated Q-value (where 'Q' denotes the quality of a taken action).

A greedy search doesn't actually guarantee a valid solution; as it could result in a terminal state with no additional moves and more than one chess piece on the board. Additionally, since there are no opposing agents, the future state of the board is fully determined by our agent's actions. As such, it is important that during training, we allow freedom for exploration.

Training the neural network works as follows:

1. Start with an initial environment which represents the board setup of a particular puzzle, s_0.
2. Perform the following steps until the game ends. A game ending is characterized by no valid moves remaining but multiple pieces exist on the board, or a single piece is present on the board).
    - The agent picks an action (i.e. which piece to move and where to move it) based on the current game state s_t.
    - The environment is updated to reflect the chosen action a_t (e.g. a piece is removed from the board and the killing piece assumes the board position of the killed piece).
    - A reward is calculated (which reflects the value of taking that action).

Each new state (2b above) is saved along with the associated reward as part of a memory structure. Such a structure represents the networks historical experiences. Next, a process called experience replay is performed on a random sample of experiences from memory.

Experience replay is used to help update the Q-value for each experience. This is done by taking the maximum q for a given action, multiplying it by a (pre-selected) discount factor, and finally adding it to the current state reward. Formally:

$$Q_{updated}(s_t, a_t) = (1 - \alpha)Q(s_t, a_t) + \alpha(reward_t + \gamma \max_{a \in Actions_{s_{t+1}}} \{Q(s_{t+1}, a)\})$$

The α variable in the formula above denotes the learning rate; γ denotes the pre-selected discount factor.

3.4.4 Exploration vs Exploitation

A purely random policy will at some point visit every state and transition many times if given enough iterations. We applied a ϵ-greedy policy such that at each time step the agent acts randomly with probability ϵ, or greedily (which selects the action with highest Q-Value) with probability 1-ϵ. Compared to a completely random policy, the ϵ-greedy policy will spend more time exploring the environment. Though Q-value estimates may improve, the agent will still have the opportunity to explore unknown regions.

3.4.5 Guaranteeing a Solution

Unfortunately, as the difficulty of a puzzle increases, it becomes less likely that our model would always choose the correct action the first time around at each state. In other words, this approach alone does not guarantee that our model will solve any puzzle on its first attempt.

To counteract such a limitation, we embedded our DRL model within a backtracking implementation where the model recommends actions and the backtracking restores the environment to a previous state upon premature game termination. As a result, the DRL acts as a pseudo action ranking model for backtracking which reduced the number of prior explored paths before reaching a correct solution (refer back to Figure 1 as an example). Such an approach will always result in an eventual correct solution. If the average number of paths explored is strictly less than that needed by the baseline model, we'd consider our proposed model to outperform the brute force approach.

Refer to Figure 2 for an excerpt from our implementation that demonstrates how we switch between policies.

The highlighted line in Figure 2 allows our Backtracking algorithm to swap between policies. Our policies were:

```
# ------------------------------------------------------------
# Generalized implementation of the Backtracking search
# algorithm using a recursive approach.
# ------------------------------------------------------------
def backtrackingSearch(problem, approach):
    # (technicality: using array because of Python scoping)
    bestTotalCost = [float('+inf')]
    bestHistory = [None]
    absoluteExplorationCount = [0]
    winningPiece = None
    winningLocation = (None, None)

    def recurse(state, history, totalCost):#, depth):
        absoluteExplorationCount[0] += 1

        # There is an issue with Python scoping of variables where
        #  if we attempted to use bestTotalCost in this function,
        #  it would actually create a new local variable and not
        #  write over the original bestTotalCost variable defined
        #  at the higher scope. To correct this issue, we simply
        #  forced the higher scope version to be a single element list

        # At |state| having undergone |history|, accumulated |totalCost|.
        #  Explore the rest of the subtree under |state|
        if problem.isEnd(state):
            # Update the best solution so far
            if totalCost < bestTotalCost[0]:
                bestTotalCost[0] = totalCost
                bestHistory[0] = history
            return

        # Recurse on children
        actions = problem.actions(state)

        rankedActions = approach.adjustActionRanking(state, actions)

        for action in rankedActions:

            newState = problem.succ(state, action)
            cost = problem.cost(newState, action)

            # 'history' is a list, we want to send a new list into our
            #  recursive step, so we use a "+" to create a new temporary list. If
            #  we use "append", our list will be modified since 'history' will
            #  overwritten otherwise (in modern literature, the details of passing
            #  by value and passing by reference are odd for Python and considered
            #  not to be applicable to Python)
            recurse(newState, history + [(action, newState, cost)], totalCost + cost)#, depth - 1)

    recurse(problem.startState(), history=[], totalCost=0)#, depth = 8)

    if not bestHistory[0] is None and len(bestHistory[0]) > 0:
        (action, newState, cost) = bestHistory[0][len(bestHistory[0])-1]
        (row, col) = (action[0][0] + action[1][0], action[0][1] + action[1][1])
        winningPiece = action[0][2]
        winningLocation = (row, col)

    return (bestTotalCost[0], bestHistory[0], absoluteExplorationCount[0], winningPiece, winningLocation)
```

Figure 2: Switching Policies

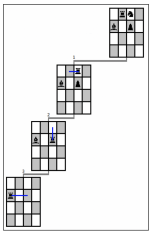

Figure 3: DRL for Puzzle #2

1. A default solution approach: performs no special ordering on the available actions at time t for state s.

2. A deep reinforcement learning approach: performs Q-value assignment.

The source code is parameterized allowing us to swap approaches for experimentation and testing. The deep reinforcement policy generated approach was able to generate the result set visually described in Figure 3 for the same puzzle described in the previous section. In other words, moving the ROOK

was ranked higher than all other actions and the model was able to complete the puzzle is far lesser moves compared to the backtracking approach.

3.4.6 Model Results

The DRL model correctly identified 100% of the correct solutions for all puzzles. Based on our evaluation criteria, this was required for the DRL model to be considered a viable model.

We graphed the average number of paths explored in Figure 4, the DRL approach consistently explored less paths than the pure backtracking approach.

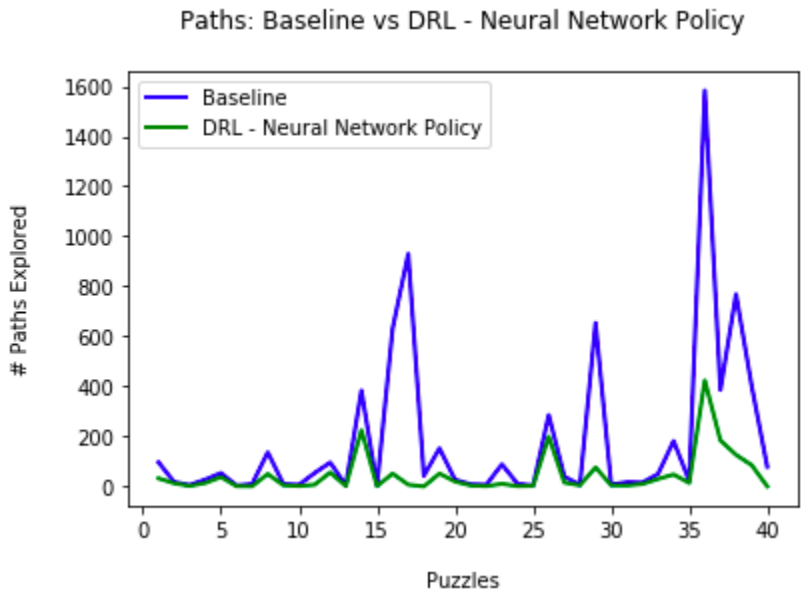

Figure 4: Prior Paths Explored between Backtracking and DRL

3.4.7 Key Observations

If our DRL approach ever degrades in its ability to estimate future values of actions, the worst thing that would occur is that the overall ranking degrades to the ranking as found in our default (no intelligence) pure backtracking algorithm (i.e. our baseline model). This effectively means that our DRL approach is bounded above by the baseline model in terms of the number of paths explored. Hence, our DRL approach is guaranteed to perform better or at the same level of the baseline model. Another key observation to note, which is a consequence of our previous observation, our DRL approach will always produce the correct target solution.

4 Literature Review

The application of deep reinforcement learning models to state-based problems is not a new concept. Some of the earliest acclaimed attempts include the work of Arthur Samuel in checking computers to play chess in the late 1960's (Sutton, 1998); as well as the development of TD-Gammon by Gerry Tesauro in the early 1990's. Some of the more modern literature on the subject matter is available from Google Brain on AlphaGo and AlphaZero. In particular, AlphaZero demonstrated the effectiveness of Tabula Rasa Monte Carlo Search which allows learning through self-play.

Majority of the literature we reviewed included multi-agent scenarios. In the absence of an actual opposing agent, there was still some form of nature, which implicitly acted as an opposing agent. In contrast, given the dimensions of the board we experimented with, our ultimate search space was relatively small compared to multi-agent games such as Chess, Pac-Man, and Atari (all other games where deep reinforcement learning was successfully applied).

5 Error Analysis

As noted earlier in the document, if our DRL approach ever degrades in its ability to estimate future values of actions, it will generate action rankings that offer no additional benefit over that of traditional backtracking (with no special logic applied). Fortunately, such a situation does not present an issue since our model does perform better or as well as the baseline model.

As an example, we illustrate the solutions of puzzle 5 from our human generated dataset; we noted that both the default backtracking model and the deep reinforcement learning model required roughly 13 steps to identify a solution.

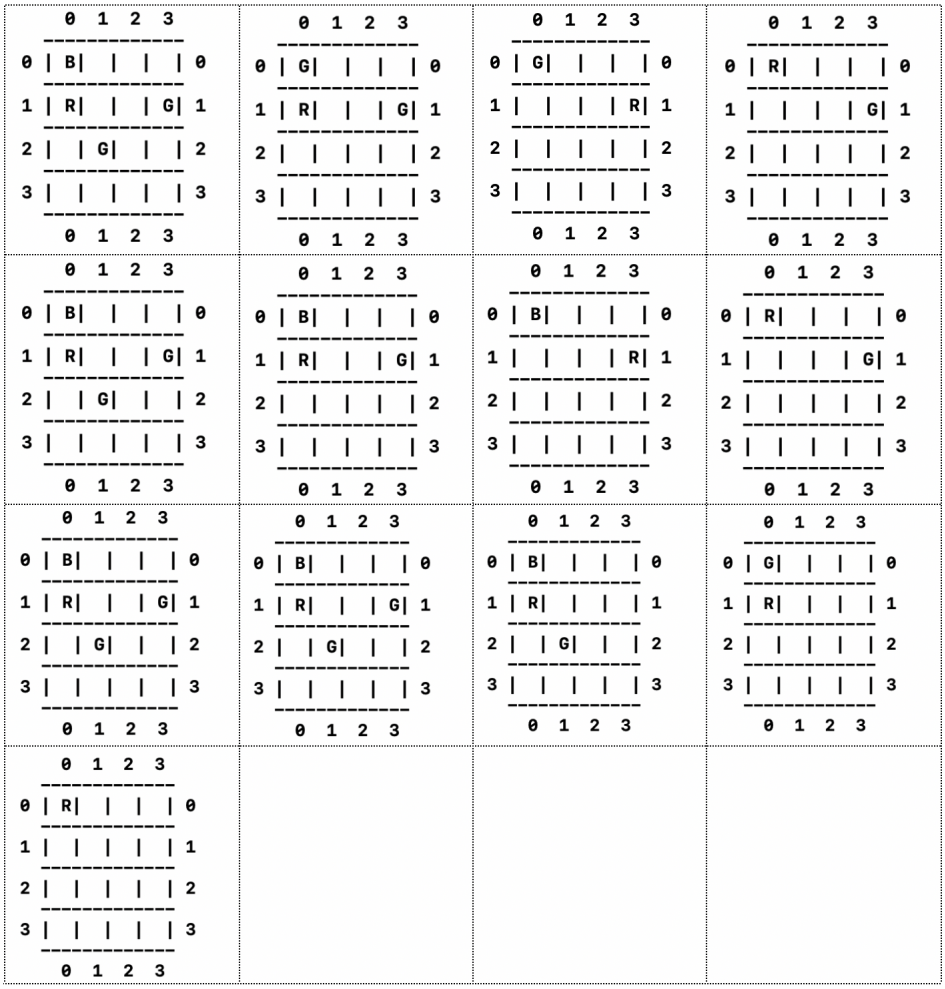

Figure 5: Tie Between Backtracking and DRL

6 Appendix

6.1 Pieces, Movements and Game Play

With Solitaire Chess, the piece movements are the same as when playing classical Chess. The player's state evaluation and strategy are also similar to thinking employed when playing classical Chess. The key differences between Solitaire Chess and classical Chess are:

- Solitaire Chess is a single-player game
- All of the pieces are the same color in Solitaire Chess

- Pawns can be placed anywhere on the board
- Pawns are not promoted, and captured pieces cannot be restored
- Kings cannot be placed into a state of check and should never be captured
- Every move must result in a piece capture

The pieces and their movement rules are as follows:

- **King**: Moves exactly one square horizontally, vertically or diagonally.
- **Queen**: Moves across any number of vacant squares horizontally, vertically or diagonally.
- **Knight**: Moves in "L" shapes (two steps in one direction, and one step in another direction at a 90-degree angle of previous direction) over vacant or occupied squares; but only the piece in the final landing square is captured.
- **Bishop**: Moves across any number of vacant squares diagonally.
- **Rook**: Moves across any number of vacant squares horizontally or vertically.
- **Pawn**: Moves upward only, one space at a time, but can move upward diagonally to capture pieces.

The following describes a typical setup and play of a game:

1. A person (known as the composer) places a subset of pieces on a checkered board (typically 4 by 4 in dimension)
2. The initial position of the pieces constitutes a specific game (or puzzle)
3. A person (known as the player) moves the pieces according to the movement rules, capturing a piece on every move until only a single piece resides on the board (at which point the game ends)

A specific game may have multiple step by step solutions but can only have a unique final board piece and its location on the board.

References

[1] Sutton, R.S. (1998) Reinforcement Learning: An Introduction. Cambridge, MA: MIT Press.