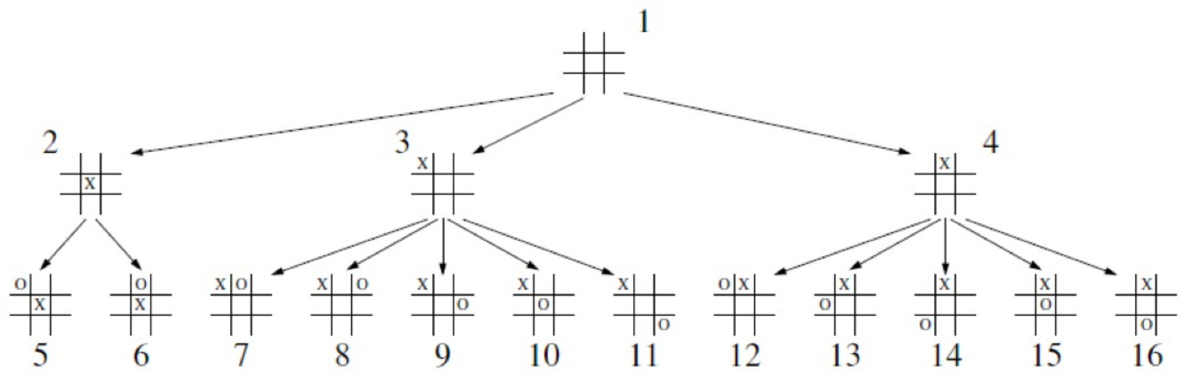
Game Trees

Introduction

In this lab, we will practice game trees which are one of the most important application in Artificial Intelligence. A game tree represents the possible choices of moves that might be made by a player or computer during a game. In these trees the root of a tree represents the initial configuration of the game, and each lower level represents all possible moves from current state (node).

For example, the following represents a partial game tree for the Tic-Tac-Toe game. The root of the tree is the initial board of the game which is empty. The nodes at level 1 are boards that result from possible moves from the board at level 0 (root). Nodes at level 2 are boards that result from possible moves from the boards at level 1. This tree show a few of the possible moves.



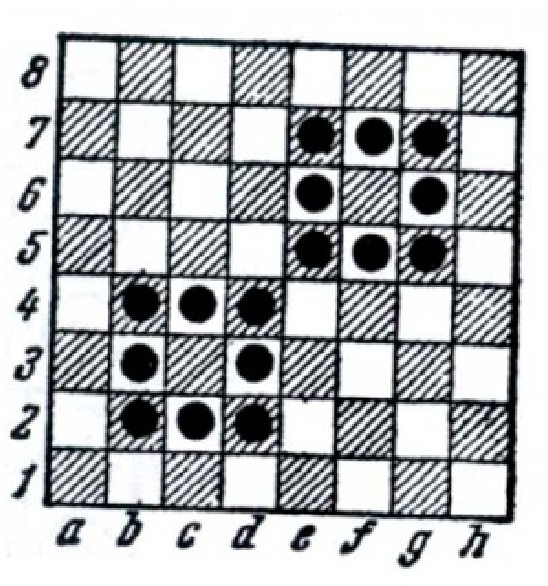
The purpose of this assignment is to help you:

1. Tree search and traversal of non-physical tree.
2. Experience game trees with boards and moves
3. Motivation for dynamic programming that we will be studying in the near future

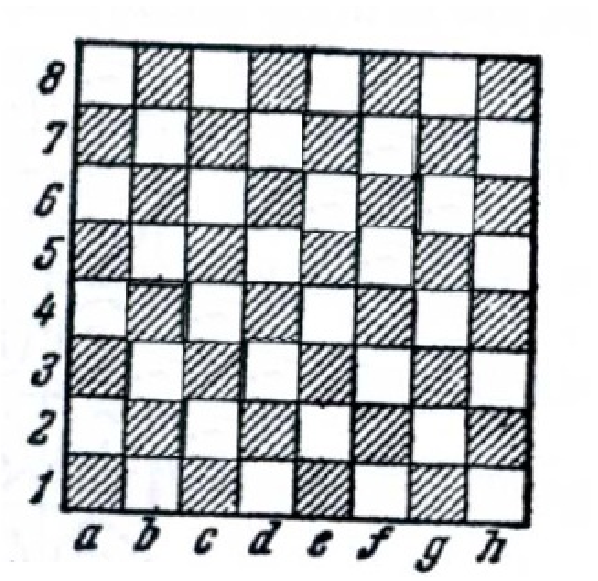
The Knight’s Move

The problem you are required to solve in this lab is called Knight’s Move. To solve this problem you don’t need to be a chess player, but rather you need to know the way a knight moves on a chessboard: two squares in one direction and one square at a right angle to the first direction or one square in one direction and two square at a right angle to the first direction. The knight's move always creates an L shape.

The following diagram shows 16 black pawns on a board. The question that your program needs to answer is whether a knight can capture all the 16 pawns in 16 moves or not. Assume we have a chess board with `k` pawns and a knight, the goal is to have the knight eat all the pawns in `k` moves. In the following image you can see the initial state and the goal state. The initial state will be the root of the tree and goal state is one possibility for a leaf in case of success.



Initial state



Goal state

In this assignment you will implement a program that uses BFS search to solve the Knight's Move problem. Your program will take an initial board, and then try to find a sequence of moves from the initial position to the goal position. The goal position is a board that can be one of two:

- if there are no pawns, then return SUCCESS
- if there are pawns, but no more knight moves that eat a pawn, then return FAIL.

However, if there are pawns and there are moves that can eat a pawn, use recursion. Once again, there are two types of outcomes:

- if all children fail, return FAIL
- if at least one succeeds, return SUCCESS.

A search tree represents the set of intermediate board states as the path-finding algorithm progresses.

Part 1 (in-lab):

In the starter code your are given the definition of a class `Board` that represents a chess board with one knight and `k` pawns. Every new board is passed a list of tuples where the first tuple represents the location of the knight on the board and all other tuples represent the locations of the pawns on the board. The `__init__` method extracts this information from the list and initializes the attributes `knight` and `pawns` accordingly.

In addition to this, the class `Board` contains the method `printBoard` that prints a board on the screen with an optional heading. The class also defines the method `findGoodMoves` that returns a list of good moves. That is, when applying each move on the knight of the current board it will yield a new location where a pawn resides. Each move is described by a tuple that represents a location on the board.

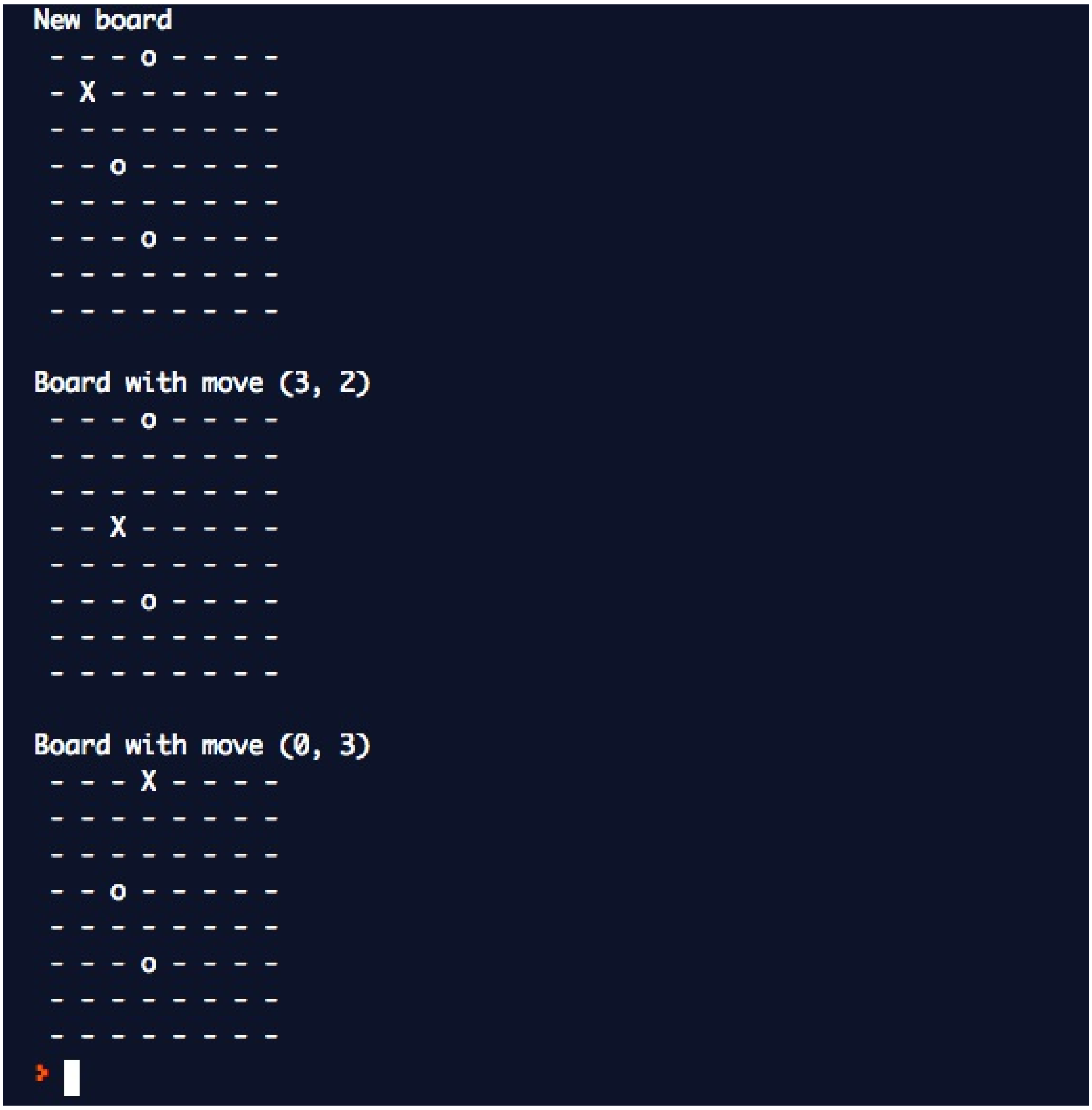
The `applyMove` method receives a move that is applied on the knight of the current board. The knight will be moved to a new location according to this move if the move is valid (invalid moves are those that go out of the chess board) and return `True`; it will return `False` otherwise. If the new location is occupied by a pawn then this pawn will be removed from the board (eaten by the knight) and this will create a new board state.

Your task:

On the basis of this class you are required to write a method called `printGoodMovesBoard`. Given a starting position, `printGoodMovesBoard` will print all the new boards that are produced from good moves (a good move is one that captures a pawn). Format your output to match the following example:

```
bb = Board([(1,1), (3,2), (5,3), (0,3)])
bb.printBoard("New board")
bb.printGoodMovesBoard()
```

Output:



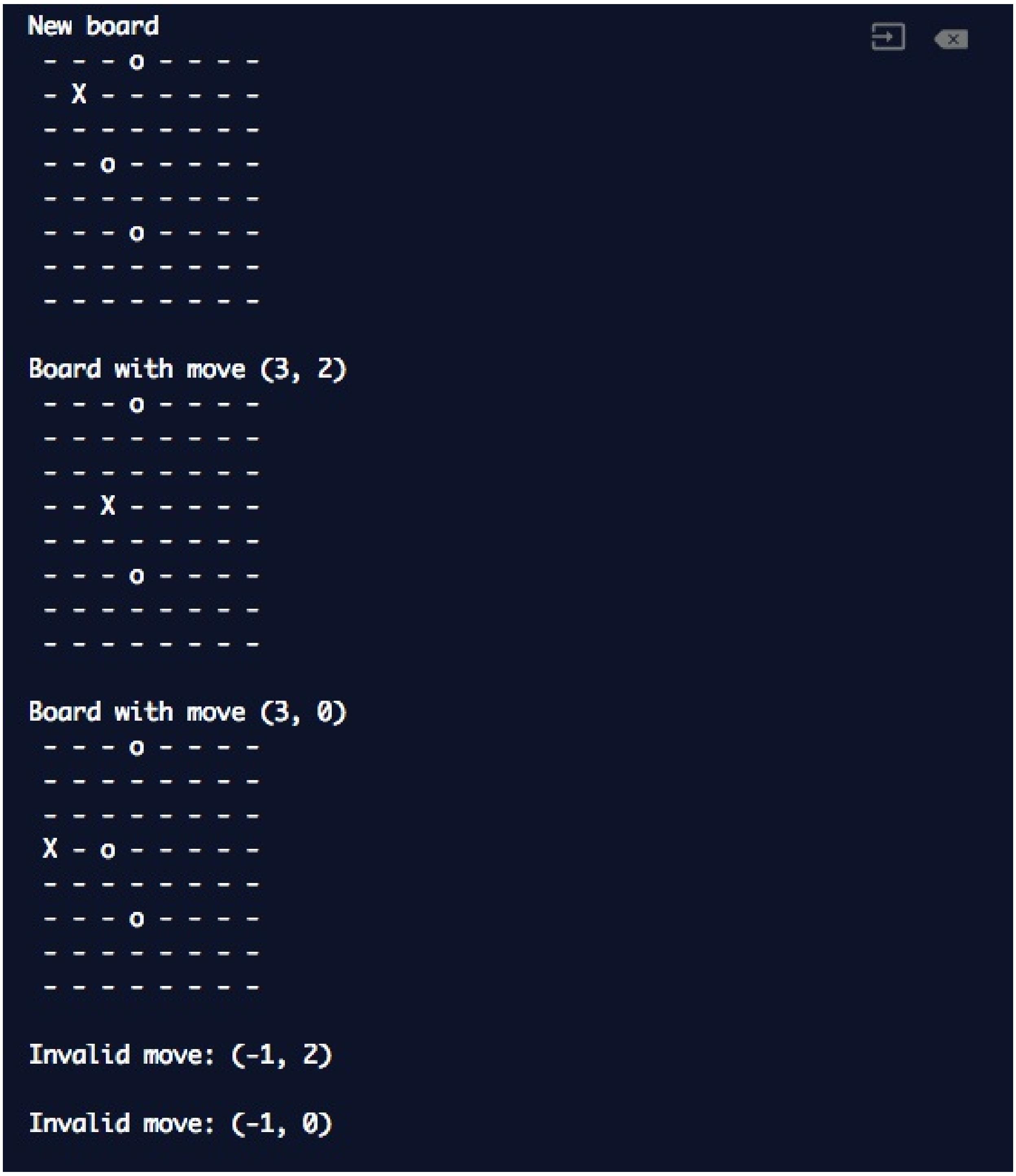
```
New board
 - - - o - - - -
 - X - - - - - -
 - - - - - - - -
 - - o - - - - -
 - - - - - - - -
 - - - o - - - -
 - - - - - - - -
 - - - - - - - -

Board with move (3, 2)
 - - - o - - - -
 - - - - - - - -
 - - - - - - - -
 - - X - - - - -
 - - - - - - - -
 - - - o - - - -
 - - - - - - - -
 - - - - - - - -

Board with move (0, 3)
 - - - X - - - -
 - - - - - - - -
 - - - - - - - -
 - - o - - - - -
 - - - - - - - -
 - - - o - - - -
 - - - - - - - -
 - - - - - - - -
>
```

Next, given a starting position for the Knight, there are at most eight moves that the Knight can make from that position. Write the method `printAllMovesBoard` prints out boards with all the valid moves (invalid moves are those that go out of the chess board). To learn about the possible moves from a given location on the board please consider the local variable `moves` in the method `findGoodMoves`. Format your output to match the following example:

```
bb = Board([(1,1), (3,2), (5,3), (0,3)])
bb.printBoard("New board")
bb.printAllMovesBoard()
```

Output:

Note: This image shows only the partial output.



```
New board
 - - - o - - - -
 - X - - - - - -
 - - - - - - - -
 - - o - - - - -
 - - - - - - - -
 - - - o - - - -
 - - - - - - - -
 - - - - - - - -

Board with move (3, 2)
 - - - o - - - -
 - - - - - - - -
 - - - - - - - -
 - - X - - - - -
 - - - - - - - -
 - - - o - - - -
 - - - - - - - -
 - - - - - - - -

Board with move (3, 0)
 - - - o - - - -
 - - - - - - - -
 - - - - - - - -
 X - o - - - - -
 - - - - - - - -
 - - - o - - - -
 - - - - - - - -
 - - - - - - - -

Invalid move: (-1, 2)

Invalid move: (-1, 0)
```

Part 2:

In this part you are required to provide a solution for the original problem. That is, given an initial board object, you need to write a function that returns `True` if it is possible to capture all the pawns on that board with `k` moves where `k` is the number of pawns on the board. Otherwise, the method will return `False`. Write two implementations of this - first using DFS and then using BFS. The functions will be called `dfsCapture` and `bfsCapture`.

Example:

```
bb = Board([(1, 1), (0, 3), (1, 5), (2, 3)])
print(dfsCapture(bb))       ## prints True
print(bfsCapture(bb))       ## also prints True
```

Hint: `dfsCapture` uses `findGoodMoves` to generate the moves and recursively builds the tree dynamcially by applying the moves. `bfsCapture` also uses `findGoodMoves` to generate the moves, but uses a queue to store the nodes (or boards) to visit and builds the tree iteratively. The base case for both is when the board has no more pawns left.

Part 3:

In this part, write a function `findPath` that improves your functions in part 2 by returning a list that is the first solution path you come across. This list will contain all the positions the knight moves to starting with the initial state and ending with the final state. In other words, the list is a tuple of all the knight positions in the solution path. If there is no path, return an empty `list`.

Example:

```
bb = Board([(1, 1), (0, 3), (1, 5), (2, 3)])
print(findPath(board))          ## prints [(2, 3), (1, 5), (0, 3)]
```

Hint: The solution is very similar to part 2. Keep track of all the knight positions and return this list as soon as you reach the base case.

Part 4:

In this last part, write a function `findAllPaths` that further improves the function in part 3 by returning all possible solutions. That is, the function should return a list of lists where each inner list is a solution path. If there are no paths, return an empty `list`.

Example:

```
bb = Board([(1, 1), (0, 3), (1, 5), (2, 3)])
print(findAllPaths(bb))
## prints [[(1, 1), (2, 3), (1, 5), (0, 3)], [(1, 1), (0, 3), (1, 5), (2, 3)]]
```