Data Science Code Series 2: Knight´s tour algorithm

Learning Goals

In this notebook, we will tackle several tasks using the Python programming language, focusing on Informed Search Trees.

By the end of this document, you will have gained the ability to:

- Define the state for a given problem.
- Express problems in the context of search trees.
- Implement the A* search strategy to discover optimal solutions for various problems.

Chess Knight Journey

Problem Description

In this notebook, we'll tackle the challenge of finding a path for a knight on a chessboard to reach a specified destination starting from a given source position. This intriguing problem is often referred to as a "knight's tour," which involves a sequence of knight moves on the chessboard. The objective is for the knight to land on each square exactly once during its journey, without revisiting any square.

Challenge: Implementing A* Search

In this exercise, we will use the A* search technique, similar to BFS, to solve the problem of finding a path from the starting position to the ending position of the knight on the chessboard. To do this, we need to define the problem in terms of the current state.

In this context, the state is represented by the knight's position on the chessboard. To manage this state, we've created an object called **Knight_state**. It's characterized by the knight's x position and y position on the chessboard. Additionally, we've included the parent node as one of its attributes.

The node's heuristic value is designated by the h attribute, while the cost from the starting node is recorded as g. The total cost, or the value of the evaluation function, is determined using the cost attribute. For the root node, the parent node is set to None, and the cost from the starting node is 0 by default, unless otherwise specified.

This setup allows us to efficiently navigate and evaluate the knight's tour on the chessboard.

Step 1

In the realm of the A* Algorithm, our trusty evaluation function is represented by the equation:

$$Cost = fn = gn + hn$$

Now, here's the task: we need to craft a method called **calc_cost()** within the **Knight_state** class. What this method does is calculate the evaluation function for a particular state.

For our heuristic value, we'll be considering the straight-line distance from the current position to the desired end position. This distance can be computed using the Euclidean distance formula. So, if we have the coordinates (x_1, y_1) for the current state and (x_2, y_2) for the end state, our heuristic value (hn) is calculated like this:

$$\text{Heuristic} = hn = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Here comes the interesting part: the function takes the goal state as its parameter. Inside this function, it's got a dual mission. First, it calculates the heuristic value (hn). Second, it computes the cost. Now, keep in mind that the value of g has already been specified when we created the object, so that part is taken care of. Once the function has done its job, it updates the values of the attributes h and cost.

Code:

In [1]:
```python
import math

# STEP 1: Define the Knight_state class

class Knight_state:
    def __init__(self, x, y, g=0, parent=None):
        """
        Initialize a Knight_state.

        Args:
        x (int): x-coordinate on the chessboard.
        y (int): y-coordinate on the chessboard.
        g (float): The cost from the starting node to this node (default is 0).
        parent (Knight_state): The parent node (default is None).

        Attributes:
        x (int): x-coordinate.
        y (int): y-coordinate.
        g (float): Cost from the starting node.
        h (float): Heuristic value.
        cost (float): Total cost.
        parent (Knight_state): Parent node.
        """
        self.x = x
        self.y = y
        self.parent = parent
        self.g = g
        self.h = None
        self.cost = None

    def calc_cost(self, goal):
        """
        Calculate the heuristic value (h) and total cost (cost) for this Knight_state.

        Args:
        goal (Knight_state): The goal state to calculate the heuristic value.

        Updates:
        h (float): Heuristic value.
        cost (float): Total cost using A* evaluation function.
        """
        # Calculate the Euclidean distance (heuristic value) from the current state to the goal state
        self.h = math.sqrt((goal.x - self.x) ** 2 + (goal.y - self.y) ** 2)

        # Calculate the total cost using the A* evaluation function
        self.cost = self.g + self.h

    def __hash__(self):
        """
        Hash function to enable storing Knight_state objects in sets and dictionaries.

        Returns:
        hash value: Hash value based on x, y, g, h, and cost attributes.
        """
        return hash((self.x, self.y, self.g, self.h, self.cost))

    def __eq__(self, other):
        """
        Compare if two Knight_state objects are equal based on their x and y coordinates.

        Args:
        other (Knight_state): Another Knight_state to compare.

        Returns:
        bool: True if x and y coordinates are equal, False otherwise.
        """
        return (self.x, self.y) == (other.x, other.y)

'''
Define the chessboard size as 8 by 8.
Our objective is to find a path from the top-left corner to the bottom-right corner.
Let's establish the start and end states.
'''

# Define the chessboard size as 8 by 8
N = 8

# Create the start and end states
start = Knight_state(7, 7)
```

```
end = Knight_state(0, 0)

# Calculate the evaluation function value for the start node
start.calc_cost(end)
print("The evaluation function value for the start node is:", start.cost)
```

```
The evaluation function value for the start node is: 9.899494936611665
```

Step 1.1

We've got two handy functions here, **isValid()** and **next_positions()**, all set up for you.

isValid() is the one to check if a given coordinate is playing by the rules or not. It's like the referee of the game.

next_positions(), on the other hand, is your playbook for the next moves. It hands you a list of coordinates for the knight's next possible positions, based on where it's currently at.

And don't forget, we've got the board size marked as **N**, globally set at 8. So, when you're ready to make some moves using these functions, just pass along that **N** as your parameter for the game board size.

Code:

In [2]:
```
# STEP 1.1: Define position validation and next positions functions.

def isValid(x, y, N):
    """
    Check if a given coordinate is within the bounds of the chessboard.

    Args:
    x (int): x-coordinate to check.
    y (int): y-coordinate to check.
    N (int): The size of the chessboard.

    Returns:
    bool: True if the coordinate is valid, False otherwise.
    """
    return not (x < 0 or y < 0 or x >= N or y >= N)

def next_positions(x, y, N):
    """
    Get a list of coordinates for the knight's next possible positions.

    Args:
    x (int): Current x-coordinate of the knight.
    y (int): Current y-coordinate of the knight.
    N (int): The size of the chessboard.

    Returns:
    list of tuple: List of valid coordinates for the knight's next moves.
    """
    next_pos = []
    row = [2, 2, -2, -2, 1, 1, -1, -1]
    col = [-1, 1, 1, -1, 2, -2, 2, -2]
    for i in range(len(row)):
        x1 = x + row[i]
        y1 = y + col[i]
        if isValid(x1, y1, N):
            next_pos.append((x1, y1))
    return next_pos
```

Step 2

Alright, let's whip up a function called **get_neighbours()**. It's got a simple job: take the current state of our knight and give us a list of possible next states, complete with their calculated costs.

Input: Feed it the current state (as a Knight_state object).

Output: You get in return a bunch of neighboring states, all bundled in a nice list of Knight_state objects.

Here's the game plan:

1) For figuring out where the knight can go next, we're tapping into the **next_positions()** function we've got on hand. That's where we get a list of possible coordinates.

2) For each of those coordinates, we're creating a fresh **Knight_state** object. These will be our neighboring states.

3) Now, let's talk costs. We've got two knights, our current one and the neighbor. To reach that neighbor, we've got a transition cost. That's what we call the "step." We calculate it like this:

$$\text{Transition cost} = \mathit{Step} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

- Where (x_1, y_1) are the coordinates of our current state, and (x_2, y_2) are the coordinates of the neighboring state.
- We also make sure that the parent node of these neighboring states is set to the current state. That's important because it helps us backtrack the path to find the solution.

4) Once these objects are all set up, we give them a quick call to the **calc_cost()** method to update their cost values.

And there you have it! We've got ourselves a bunch of potential moves to explore.

Code:

In [3]:
```
# STEP 2: Define the function to input th knight's current position and output a list of the possible
# next positions, by cost.

def get_neighbours(curr_state, N):
    """
    Get a list of neighboring states with calculated costs for the knight's next moves.

    Args:
    curr_state (Knight_state): The current state of the knight.
    N (int): The size of the chessboard.

    Returns:
    list of Knight_state: List of neighboring states with updated costs.
    """
    neighbors = []

    # Get the current state's coordinates
    x1, y1 = curr_state.x, curr_state.y

    # Get the next possible coordinates using next_positions()
    next_coords = next_positions(x1, y1, N)

    for x2, y2 in next_coords:
        # Calculate the cost for the transition (step)
        step = math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)

        # Create a Knight_state object for the neighboring state
        neighbor_state = Knight_state(x2, y2, g=curr_state.g + step, parent=curr_state)

        # Calculate the cost and heuristic value for the neighboring state
        neighbor_state.calc_cost(end)  # Assuming 'end' is the goal state

        # Append the neighboring state to the list
        neighbors.append(neighbor_state)

    return neighbors

# Output next positions information
neighbour_pos = get_neighbours(start, N)
next_state = neighbour_pos
print("Number of the possible next positions:",len(next_state))
print()
print("One of the possible next positions description:")
print("Coordinates:", (next_state[0].x, next_state[0].y))
print("Cost:", next_state[0].cost)
```

```
Number of the possible next positions: 2

One of the possible next positions description:
Coordinates: (5, 6)
Cost: 10.046317653406444
```

Step 3

Time to craft a function called **get_min()**. Its job is simple: you feed it a list of knight states, and it's going to tell you which knight state has the minimum cost, and it'll give you the index too.

Input: Just hand it a list of states or nodes. And in this list, we're expecting those trusty Knight_state objects. *(List of Knight_state object)*

Output: What you'll get in return is the index of that knight state with the minimum cost. It's just an integer that points you right to the winner. *(Integer)*

Code:

In [4]:
```
### STEP 3: Define a function that outputs the minimum cost of the different next possible knight's positior

def get_min(frontier):
    """
    Find the index of the knight state with the minimum cost in a list of states.

    Args:
    frontier (list of Knight_state): List of knight states to search for the minimum cost.

    Returns:
    int: The index of the knight state with the minimum cost.
    """
    min_state = min(frontier, key=lambda state: state.cost)

    # Find the index of the minimum state in the list
    min_index = frontier.index(min_state)

    return min_index

# Function call and necessary info outputs
n = Knight_state(6,5)
list_states = get_neighbours(n,N)
min_i = get_min(list_states)
cost_list = [i.cost for i in list_states]
pos_list = [(i.x,i.y) for i in list_states]
print("The list of costs:", cost_list)
print("Index of minimum cost:", min_i)
print("Minimum cost:",cost_list[min_i])
```

```
The list of costs: [9.447170528427769, 7.89292222699217, 12.135562914111455, 9.851841083363698, 10.83839324
4542416, 8.06701987234509]
Index of minimum cost: 1
Minimum cost: 7.89292222699217
```

Step 3.1

In this next step of our Knight's Tour quest, we're gearing up with some essential gear. First, we've got our trusty "frontier" list, where we'll store the states we're about to explore. Then, there's the "explored" list, keeping a record of the places we've already been. We kick things off by adding our starting position to the frontier – it's like planting our flag and saying, "Let the adventure begin!" We've also got the "goal" waiting in the wings to store the end state once we've conquered it. To keep things organized, we've got our "itr," which counts our steps, and "min_index," the ace up our sleeve for spotting the state with the minimum cost. These tools are our navigation aids as we embark on the journey to solve the Knight's Tour puzzle.

In [5]:
```
# Initialize an empty list to store states to be explored (frontier) and explored states.
frontier = []
explored = []

# Add the starting state (start) to the frontier, as we begin the search from the starting position.
frontier.append(start)

# Initialize the goal variable, which will hold the goal state when found.
goal = None

# Initialize the iteration counter to keep track of the number of iterations in the search.
itr = 1

# Initialize the minimum index to keep track of the state with the minimum cost in the frontier.
min_index = 0
```

Step 4

Alright, let's dive into the iteration part. We're going to cruise through the queue, hitting up different states in our tree. Here's what's on the itinerary:

1) We start by popping out the element from the frontier list. But we're picky; we want the one with the minimum cost, and we know where to find it – **min_index**.

NOTE: At the beginning of this journey, we've only got the start node. So, initially, **min_index** is set to 0.

Once we've got that node, we do a quick check. Are we at our destination, our **end state**? If we are, we've reached our **goal**.

If not there yet, it's time to make some new friends. We find the **neighbors**, and if we haven't **explored** them before and they're not already in the **frontier**, we add them. But, if they're already in our queue, we update their state.

So, we're popping nodes, checking if we're done, and making new friends along the way. That's how we roll through this tree.

Code:

In [6]:
```python
### STEP 4: Create the iterative algorithm to look for the best path for the knight.
while len(frontier)>0:
    node = None

    '''

    Sub-Task_1: Pop the Knight state from the frontier using the minimum index.
                Update the 'visited' and 'explored' attributes if needed.

    Sub-Task_2: Check if the current node is the end node.
                If it is, assign the node to the 'goal' variable.
                Terminate the loop.

    '''

    min_index = get_min(frontier)
    node = frontier.pop(min_index)
    explored.append(node)

    if node.x == end.x and node.y == end.y:
        goal = node
        break

    neighbours = get_neighbours(node, N)

    '''

    UPDATING FRONTIER:
        - Iterate through neighboring states.
        - Add a state to the frontier if it hasn't been explored and is not already present in the frontier.
        - If the neighboring state is already in the frontier, update the node in the frontier if the cost
          is lower than the previous cost.

    '''
    for neighbour in neighbours:
        if (neighbour not in frontier) and (neighbour not in explored):
            frontier.append(neighbour)
        elif neighbour in frontier:
            for i in range(len(frontier)):
                if neighbour == frontier[i]:
                    if neighbour.cost < frontier[i].cost:
                        frontier[i]=neighbour


    if len(frontier)>0:
        min_index = get_min(frontier)

    itr += 1
    if len(frontier)==0:
        print("Solution not found")
```

Step 4.1

The assert statement below, serves as a safeguard. Its purpose is to ensure that the variable goal is not empty or undefined before proceeding further in the code. In technical terms, it acts as a sanity check, preventing the execution of subsequent code if goal happens to be uninitialized or lacking a valid value. Essentially, it's a way to catch potential errors early in the process, providing users with a helpful safety net to avoid unexpected issues. This assert statement is thoughtfully included in the notebook to maintain code reliability and stability.

In [7]:
```
assert goal is not None
```

Step 5

backtrace(node)

It's a handy function already defined for your convenience.

This function, is a critical tool that helps users trace their steps from a goal node back to the starting point. It's particularly useful in search algorithms.

Here's how it works: When called with a node as input, it examines if the node has a parent. If it's the start node (i.e., it has no parent), it simply records its coordinates and returns that as the path. If it's not the start node, it delves into the recursion magic.

It backtracks through the parent nodes, building the path incrementally by collecting the coordinates of each node visited. This way, it constructs a complete path from the goal back to the starting point. Once you call it with the goal node, it does its thing and prints out the solution path, showing you the journey from start to finish.

In [8]:
```
def backtrace(node):
    # Initialize an empty list to store the path.
    path = []

    # If the current node has no parent, it is the start node.
    if node.parent is None:
        position = (node.x, node.y)  # Get the coordinates of the current node.
        path.append(position)        # Append the coordinates to the path list.
        return path                 # Return the path.

    else:
        # Recursively backtrack through the parent nodes.
        path = backtrace(node.parent)

        position = (node.x, node.y)  # Get the coordinates of the current node.
        path.append(position)        # Append the coordinates to the path list.
        return path                 # Return the updated path.

# Example usage:
path = backtrace(goal)              # Call the backtrace function starting from the goal node.
print("The solution Path is: ")
print(path)                         # Print the resulting path.
```

```
The solution Path is:
[(7, 7), (5, 6), (4, 4), (2, 3), (0, 2), (2, 1), (0, 0)]
```

Well done!

Now, if you like, you could re-create this path either digitally or physically using these coordinates. You could also play along and try different start and end nodes out of curiosity.

Summary and Closing Notes

In this code walkthrough, we delved into the realm of informed search trees and the A *search algorithm to tackle the intriguing problem of the knight's tour on a chessboard. By defining the problem in terms of states, calculating costs using an evaluation function, and efficiently exploring neighboring states, we were able to discover the optimal path for the knight to visit every square on the chessboard without revisiting any. The A* search strategy, combined with well-crafted functions and careful exploration, led us to a satisfying solution. This exercise serves as a great example of problem-solving and algorithmic thinking in the context of a classic chess puzzle.

Credits:

By curlypetrol, 2023