Python Library Imports

```
# Imports - Libraries
import time
from typing import List # Used for storing grid (3x3)
from prettytable import PrettyTable
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import queue # Used as Fringe (data-structure for storing explored states)
```

Python Four Knights Source Code

```
class FourKnights():

  # Default constructor of 4Knights' class
  def __init__(self, start_state, goal_state):

    # Initialize start and goal state when object of class is created
    self.start_knight_state = start_state
    self.goal_knight_state = goal_state

    # Initialize list of successor states
    self.successors = list()


    #for n in range(0, len(self.start_knight_state), 3):
    #self.state_space.append(self.start_knight_state[n:n+3])

    # Define Knights Actions
    # Create all possible actions which Knights can take based on the Knight's 'L' movement rule of:
    # 2 across 1 up (2,1),(-2,1)
    # 2 across 1 down (2,-1),(-2,-1)
    # 1 across 2 up (1,2),(-1,2)
    # 1 across 2 down (1,-2),(-1,-2)
    self.knight_actions = [(2,1),(-2,1),(2,-1),(-2,-1),(1,2),(-1,2),(1,-2),(-1,-2)]


  # Successor Function - returns possible knight states which can be explored from current state within the confines of the 3x3 grid
  def successor_func(self, state):
    successor_positions = []

    # Because it's a 3x3 grid, we need to take every 3 values from state_knight_state and generate a 3x3 (row, col) list
    # 1,0,1
    # 0,0,0
    # 2,0,1
    state_space = list(list(state[n:n+3]) for n in range(0, len(state), 3))

    # Iterate over row & columns. Since grid is 3x3 we specify range == 0,1,2 (i.e. up to 3rd index)
    # [0,0], [0,1], [0,2] => row 0
    # [1,0], [1,1], [1,2] => row 1
    # [2,0], [2,1], [2,2] => row 2

    for row in range(0, 3):
      for column in range (0, 3):

        # In order to return possible knight state which can be explored (i.e. move to), we need to check if we can currently on a knight's po
        # Because the state positions without knights have been set to non-zero we use != 0 for the evaluation
        if state_space[row][column] != 0:
          # Once on a Knight's position, determine what actions can be taken
          for action in self.knight_actions:
            # Calculate next grid state space using the possible 'L' actions
            # Indices 0 & 1 based on tuple structure for knight_actions
            # _row = 0 + 2 = 2, _column = 0 + 1 = 1
            _row, _column  = row + action[0], column + action[1]

            # Because the next grid state space calculation will could be greater than 3 and the grid position values are: 0, 1 and 2 we need
            # AND we want to only move the knight to a grid position (state) which has a value of 0 (indicating no knights present), i.e. sel
            if (0 <= _row < 3 and 0 <= _column < 3) and (state_space[_row][_column] == 0):

              successor_state = [each_row.copy() for each_row in state_space]

              # Move knights position by assigning current state (based on row, column) to newly calculated grid position which is empty
```

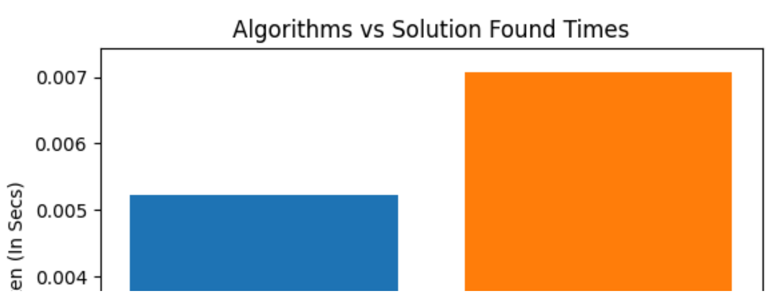
```
              # Move knights position by assigning current state (based on row, column) to newly calculated grid position which is empty
              successor_state[_row][_column], successor_state[row][column] = successor_state[row][column], successor_state[_row][_column]

              # Add new successor state to list of successors to be used for knowning path traversal to goal in the end
              # Convert knight_new_grid_state from being 2D (rows and colums) to 1D tuple list [(,),(,)] before adding.
              successor_positions.append(tuple(sum(successor_state, [])))

    return successor_positions


  # Heuristic Function - calculates the cost from a given state to the goal state
  def heuristic_func(self, current_knight_state):

    # The zip object yields n-length tuples, where n is the number of iterables passed as positional arguments to zip()
    # Iterate over each pair of generated tuples (x, y) where x == start and y == goal
    # Calculate the absolute difference between x and y
    # Sum the generated absolute values
    return sum(abs(x - y) for x, y in zip(current_knight_state, self.goal_knight_state))


  # Expand Function - obtain successors based on current_state
  # Calculate cost to successor state
  # Add successor to queue data-structure
  def expand_func(self, algorithm, fringe, current_state, parent_state, cost_to_state):

    # Start time defined
    start_time = time.time()

    # While the PriorityQueue is not empty, check if state == goal state, else get sucessor states (i.e. expand)
    while fringe.qsize() != 0:

      # .get() returns the priority & state at front of queue (FIFO based on priority cost - least cost)
      priority_cost, current_state = fringe.get()

      # Success! Goal State Obtained!!!
      if current_state == self.goal_knight_state:

          # Great, let's retrieve the path which led to the current state
          path_to_goal_state = list()

          # Retrace each state from current state to start_state where value was None
          while current_state is not None:
            # Add current_state to path_to_goal_state
            path_to_goal_state.append(current_state)

            # Assign current_state to
            current_state = parent_state[current_state]

          # Flip the list order to read from start to goal states instead of goal to start states
          path_to_goal_state.reverse()

          # Num of states explored to get to goal_state
          num_of_states_exp_to_goal = len(path_to_goal_state) - 1

          # End time defined
          end_time = time.time()

          total_time = end_time - start_time

          # Return path to goal state & number of states needed to get to goal_state
          return path_to_goal_state, num_of_states_exp_to_goal, total_time

      # Else get states which we can expand next based on cost
      else:

        if "astar" in algorithm:

          for successor in self.successor_func(current_state):
            cost_to_successor = cost_to_state[current_state] + 1

            if successor not in cost_to_state or cost_to_successor < cost_to_state[successor]:
                  cost_to_state[successor] = cost_to_successor
                  # The total priority is the cost to reach the successor state and the heuristic value of the successor state
                  total_priority = cost_to_successor + self.heuristic_func(successor)
                  # Add the successor state to the queue
                  fringe.put((total_priority, successor))
                  # The parent state of a state is the state from which the current state was reached
                  parent_state[successor] = current_state
```

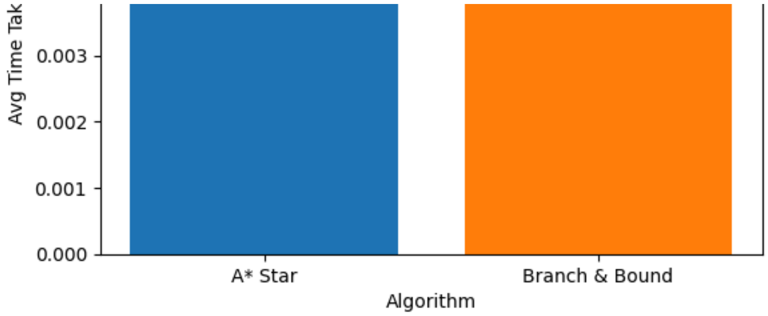
```
        elif "branch" in algorithm:

          for successor in self.successor_func(current_state):
            cost_to_successor = cost_to_state[current_state] + 1

            if successor not in cost_to_state or cost_to_successor < cost_to_state[successor]:
                  cost_to_state[successor] = cost_to_successor
                  # The total priority is the cost to reach the successor state without the heuristic value of the successor state as Branch &
                  total_priority = cost_to_successor
                  # Add the successor state to the queue
                  fringe.put((total_priority, successor))
                  # The parent state of a state is the state from which the current state was reached
                  parent_state[successor] = current_state


    # If fringe is empty then no states to explored/expanded
    # No states explored
    return None, None, 0

  # A* Search Algorithm (f(n) = g(n) + h(n), where g(n) is the cost between states and h(n) is the heuristic cost from state to goal) is used
  def astar_func(self, start_state):

    # Data structure for temporary storing states to be expanded.
    # PriorityQueue because first element indicates cost f(n) needed to reach state
    fringe = queue.PriorityQueue()

    # Add starting state to PriorityQueue
    fringe.put((0, start_state))

    # Initialize cost_to_state
    # Using a dictionary to store to utilize the key,value functionality
    cost_to_state = {start_state: 0}

    # Because we will have a go back propagation once we reach the goal state to know the path to the goal, we will need to keep track of the
    # Initialize to none because the start_state has not parent as no moves have been done
    parent_state =  {start_state: None}

    # Gets path to goal solution, number of states expanded & total time taken for search
    goal_path, total_states_exp_to_goal, total_time = self.expand_func("astar", fringe, start_state, parent_state, cost_to_state)

    return goal_path, total_states_exp_to_goal, total_time

  # Branch & Bound Search Algorithm the cost between states to goal is used to determine path traversal.
  # We are using the same Least Cost Search for Branch & Bound to best be able to compare against the A* Search
  def brand_bound_func(self, start_state):

    # Data structure for temporary storing states to be expanded.
    # PriorityQueue because first element indicates cost needed to reach state
    fringe = queue.PriorityQueue()

    # Add starting state to PriorityQueue
    fringe.put((0, start_state))

    # Initialize cost_to_state
    # Using a dictionary to store to utilize the key,value functionality
    cost_to_state = {start_state: 0}

    # Because we will have a go back propagation once we reach the goal state to know the path to the goal, we will need to keep track of the
    # Initialize to none because the start_state has not parent as no moves have been done
    parent_state =  {start_state: None}

    # Gets path to goal solution, number of states expanded & total time taken for search
    goal_path, total_states_exp_to_goal, total_time = self.expand_func("branch", fringe, start_state, parent_state, cost_to_state)

    return goal_path, total_states_exp_to_goal, total_time


def main():

  # Specify start state
  start_state = (2,0,2,0,0,0,4,0,4)

  # Specify goal state
  goal_state = (4,0,4,0,0,0,2,0,2)

  # Create object of class
```

```
    knight = FourKnights(start_state, goal_state)

    # Create a PrettyTable object
    table = PrettyTable()

    # Track average times
    astar = list()
    bnb = list()

    # Printing results
    print("4 Knights Puzzle Sample Runs:\n")

    # Measuring Averages
    # Run search algorithms (x10 times) then calculate the avgs
    for n in range(10):

      # Run A* Search Algorithm
      astar_solu_path, astar_states_val, exp_time = knight.astar_func(start_state)

      # Add times to list for calculating avg time to solution
      astar.append(exp_time)

      # Run A* Search Algorithm
      bnb_solu_path, bnb_states_val, _exp_time = knight.brand_bound_func(start_state)

      # Add times to list for calculating avg time to solution
      bnb.append(_exp_time)

      print(f"Round {n}:")

      print(f"A* States: {astar_solu_path}")
      print(f"Branch & Bound States: {bnb_solu_path}\n")

      # Add rows
      table.add_row(["A* Search", astar_states_val, exp_time, n])
      table.add_row(["Branch & Bound Search", bnb_states_val, _exp_time, n])

    # Calculate average time to solution/goal per algorithm
    avg_astar = (sum(astar) / 10)
    avg_bnb = (sum(bnb) / 10)

    # Calculate and then output avg times for A* Search vs Branch and Bound Search
    print(f"A* Search found a solution within {avg_astar} secs on average.")
    print(f"Branch & Bound Search found a solution within {avg_bnb} secs on average.\n")

    # Define the column names
    table.field_names = ["Algorithm", "Time Complexity (No. of Moves)", "Optimality (In Seconds)", "Execution Round"]

    # Print the table Header Label
    print("Table Results Summary:")

    # Print the table
    print(table)

    # Create bar chart using extracted values from table
    plt.bar("A* Star", avg_astar)
    plt.bar("Branch & Bound", avg_bnb)

    # Define chart labels
    plt.title("Algorithms vs Solution Found Times")
    plt.xlabel("Algorithm")
    plt.ylabel("Avg Time Taken (In Secs)")

    print("\n")

    # Show plot
    plt.show()


  if __name__ == '__main__':
      main()
```

```
4 Knights Puzzle Sample Runs:

Round 0:
A* States: [(2, 0, 2, 0, 0, 0, 4, 0, 4), (2, 0, 0, 0, 0, 0, 4, 2, 4), (2, 0, 0, 4, 0, 0, 4, 2, 0), (2, 0, 4, 0, 0, 0, 4, 2, 0), (2, 4, 4
Branch & Bound States: [(2, 0, 2, 0, 0, 0, 4, 0, 4), (0, 0, 2, 0, 0, 2, 4, 0, 4), (0, 0, 0, 0, 0, 2, 4, 2, 4), (0, 4, 0, 0, 0, 2, 0, 2,

Round 1:
A* States: [(2, 0, 2, 0, 0, 0, 4, 0, 4), (2, 0, 0, 0, 0, 0, 4, 2, 4), (2, 0, 0, 4, 0, 0, 4, 2, 0), (2, 0, 4, 0, 0, 0, 4, 2, 0), (2, 4, 4
Branch & Bound States: [(2, 0, 2, 0, 0, 0, 4, 0, 4), (0, 0, 2, 0, 0, 2, 4, 0, 4), (0, 0, 0, 0, 0, 2, 4, 2, 4), (0, 4, 0, 0, 0, 2, 0, 2,

Round 2:
A* States: [(2, 0, 2, 0, 0, 0, 4, 0, 4), (2, 0, 0, 0, 0, 0, 4, 2, 4), (2, 0, 0, 4, 0, 0, 4, 2, 0), (2, 0, 4, 0, 0, 0, 4, 2, 0), (2, 4, 4
Branch & Bound States: [(2, 0, 2, 0, 0, 0, 4, 0, 4), (0, 0, 2, 0, 0, 2, 4, 0, 4), (0, 0, 0, 0, 0, 2, 4, 2, 4), (0, 4, 0, 0, 0, 2, 0, 2,

Round 3:
A* States: [(2, 0, 2, 0, 0, 0, 4, 0, 4), (2, 0, 0, 0, 0, 0, 4, 2, 4), (2, 0, 0, 4, 0, 0, 4, 2, 0), (2, 0, 4, 0, 0, 0, 4, 2, 0), (2, 4, 4
Branch & Bound States: [(2, 0, 2, 0, 0, 0, 4, 0, 4), (0, 0, 2, 0, 0, 2, 4, 0, 4), (0, 0, 0, 0, 0, 2, 4, 2, 4), (0, 4, 0, 0, 0, 2, 0, 2,

Round 4:
A* States: [(2, 0, 2, 0, 0, 0, 4, 0, 4), (2, 0, 0, 0, 0, 0, 4, 2, 4), (2, 0, 0, 4, 0, 0, 4, 2, 0), (2, 0, 4, 0, 0, 0, 4, 2, 0), (2, 4, 4
Branch & Bound States: [(2, 0, 2, 0, 0, 0, 4, 0, 4), (0, 0, 2, 0, 0, 2, 4, 0, 4), (0, 0, 0, 0, 0, 2, 4, 2, 4), (0, 4, 0, 0, 0, 2, 0, 2,

Round 5:
A* States: [(2, 0, 2, 0, 0, 0, 4, 0, 4), (2, 0, 0, 0, 0, 0, 4, 2, 4), (2, 0, 0, 4, 0, 0, 4, 2, 0), (2, 0, 4, 0, 0, 0, 4, 2, 0), (2, 4, 4
Branch & Bound States: [(2, 0, 2, 0, 0, 0, 4, 0, 4), (0, 0, 2, 0, 0, 2, 4, 0, 4), (0, 0, 0, 0, 0, 2, 4, 2, 4), (0, 4, 0, 0, 0, 2, 0, 2,

Round 6:
A* States: [(2, 0, 2, 0, 0, 0, 4, 0, 4), (2, 0, 0, 0, 0, 0, 4, 2, 4), (2, 0, 0, 4, 0, 0, 4, 2, 0), (2, 0, 4, 0, 0, 0, 4, 2, 0), (2, 4, 4
Branch & Bound States: [(2, 0, 2, 0, 0, 0, 4, 0, 4), (0, 0, 2, 0, 0, 2, 4, 0, 4), (0, 0, 0, 0, 0, 2, 4, 2, 4), (0, 4, 0, 0, 0, 2, 0, 2,

Round 7:
A* States: [(2, 0, 2, 0, 0, 0, 4, 0, 4), (2, 0, 0, 0, 0, 0, 4, 2, 4), (2, 0, 0, 4, 0, 0, 4, 2, 0), (2, 0, 4, 0, 0, 0, 4, 2, 0), (2, 4, 4
Branch & Bound States: [(2, 0, 2, 0, 0, 0, 4, 0, 4), (0, 0, 2, 0, 0, 2, 4, 0, 4), (0, 0, 0, 0, 0, 2, 4, 2, 4), (0, 4, 0, 0, 0, 2, 0, 2,

Round 8:
A* States: [(2, 0, 2, 0, 0, 0, 4, 0, 4), (2, 0, 0, 0, 0, 0, 4, 2, 4), (2, 0, 0, 4, 0, 0, 4, 2, 0), (2, 0, 4, 0, 0, 0, 4, 2, 0), (2, 4, 4
Branch & Bound States: [(2, 0, 2, 0, 0, 0, 4, 0, 4), (0, 0, 2, 0, 0, 2, 4, 0, 4), (0, 0, 0, 0, 0, 2, 4, 2, 4), (0, 4, 0, 0, 0, 2, 0, 2,

Round 9:
A* States: [(2, 0, 2, 0, 0, 0, 4, 0, 4), (2, 0, 0, 0, 0, 0, 4, 2, 4), (2, 0, 0, 4, 0, 0, 4, 2, 0), (2, 0, 4, 0, 0, 0, 4, 2, 0), (2, 4, 4
Branch & Bound States: [(2, 0, 2, 0, 0, 0, 4, 0, 4), (0, 0, 2, 0, 0, 2, 4, 0, 4), (0, 0, 0, 0, 0, 2, 4, 2, 4), (0, 4, 0, 0, 0, 2, 0, 2,

A* Search found a solution within 0.005230927467346191 secs on average.
Branch & Bound Search found a solution within 0.0070782661437988285 secs on average.

Table Results Summary:
```

Algorithm	Time Complexity (No. of Moves)	Optimality (In Seconds)	Execution Round
A* Search	16	0.005034923553466797	0
Branch & Bound Search	16	0.006459951400756836	0
A* Search	16	0.004889488220214844	1
Branch & Bound Search	16	0.007148027420043945	1
A* Search	16	0.006794929504394531	2
Branch & Bound Search	16	0.0057904720306396484	2
A* Search	16	0.004841804504394531	3
Branch & Bound Search	16	0.008626461029052734	3
A* Search	16	0.004842519760131836	4
Branch & Bound Search	16	0.0066890716552734375	4
A* Search	16	0.004723310470581055	5
Branch & Bound Search	16	0.007296562194824219	5
A* Search	16	0.004757881164550781	6
Branch & Bound Search	16	0.007069110870361328	6
A* Search	16	0.006796121597290039	7
Branch & Bound Search	16	0.007210969924926758	7
A* Search	16	0.004786252975463867	8
Branch & Bound Search	16	0.007363557815551758	8
A* Search	16	0.004842042922973633	9
Branch & Bound Search	16	0.007128477096557617	9




Avg Time Tak
0.003
0.002
0.001
0.000
A* Star
Branch & Bound
Algorithm