

# A\* Algorithm Python

April 13, 2024

## Anonymous

Design a python program to implement the A\* algorithm, an informed search algorithm used for pathfinding. The program should efficiently find the optimal path from a given start point to a goal point on a grid-based environment, considering both the heuristic function and the cost function

## ChatGPT

Sure, implementing the A\* algorithm in Python is a great exercise! Here's a basic implementation:

python

```
import heapq

class Node:
    def __init__(self, state, parent=None, g=0, h=0):
        self.state = state
        self.parent = parent
        self.g = g # Cost from start node to current node
        self.h = h # Heuristic cost from current node to goal node

    def f(self):
        return self.g + self.h

class AStar:
    def __init__(self, grid, start, goal):
        self.grid = grid
        self.start = start
        self.goal = goal
        self.open_set = []
        heapq.heappush(self.open_set, (0, start)) # Priority queue
        self.came_from = {}
        self.g_scores = {start: 0}
        self.f_scores = {start: self.heuristic(start)}

    def heuristic(self, node):
        # Euclidean distance heuristic
        return ((node[0] - self.goal[0]) ** 2 + (node[1] - self.goal[1]) ** 2) ** 0.5

    def neighbors(self, node):
        x, y = node
        neighbors = [(x-1, y), (x+1, y), (x, y-1), (x, y+1)] # Assuming 4-connected grid
        valid_neighbors = [(nx, ny) for nx, ny in neighbors if 0 <= nx < len(self.grid) and
0 <= ny < len(self.grid[0]) and self.grid[nx][ny] != '#']
        return valid_neighbors

    def reconstruct_path(self, current):
        path = []
        while current in self.came_from:
            path.append(current)
            current = self.came_from[current]
        return path[::-1]

    def search(self):
        while self.open_set:
            _, current = heapq.heappop(self.open_set)
            if current == self.goal:
                return self.reconstruct_path(current)
            for neighbor in self.neighbors(current):
                tentative_g = self.g_scores[current] + 1 # Assuming uniform cost for
```

```

simplicity
    if neighbor not in self.g_scores or tentative_g < self.g_scores[neighbor]:
        self.came_from[neighbor] = current
        self.g_scores[neighbor] = tentative_g
        self.f_scores[neighbor] = tentative_g + self.heuristic(neighbor)
        heapq.heappush(self.open_set, (self.f_scores[neighbor], neighbor))

    return None

# Example usage:
grid = [
    ['.','.','.','.','.','.'],
    ['.','#','.','#','.','#'],
    ['.','.','.','.','.'],
    ['#','.','#','#','.','#'],
    ['.','.','.','.','.']
]
start = (0, 0)
goal = (4, 5)
astar = AStar(grid, start, goal)
path = astar.search()
print(path)

```

This code defines a `Node` class to represent each cell in the grid, and an `AStar` class to perform the A\* search. The `heuristic` function calculates the heuristic (Euclidean distance) from a given node to the goal. The `search` function implements the A\* algorithm, and the `reconstruct_path` function reconstructs the optimal path once the goal is reached. Finally, an example usage demonstrates how to use the AStar class to find the optimal path on a grid.

### Anonymous

Design a Python program to solve Sudoku puzzles using heuristic search strategies and assuming the static ordering of cells based on suitable constraints

### ChatGPT

Certainly! Here's a Python program using the A\* algorithm combined with heuristic search strategies to solve Sudoku puzzles:

python

```

class SudokuSolver:
    def __init__(self, puzzle):
        self.puzzle = puzzle
        self.size = len(puzzle)
        self.box_size = int(self.size ** 0.5)

    def is_valid(self, row, col, num):
        # Check if the number is already in the current row
        if num in self.puzzle[row]:
            return False

        # Check if the number is already in the current column
        if num in [self.puzzle[i][col] for i in range(self.size)]:
            return False

        # Check if the number is already in the current box
        start_row, start_col = self.box_size * (row // self.box_size), self.box_size * (col // self.box_size)
        for i in range(start_row, start_row + self.box_size):
            for j in range(start_col, start_col + self.box_size):
                if self.puzzle[i][j] == num:
                    return False

```

```

        return True

    def find_empty_cell(self):
        for i in range(self.size):
            for j in range(self.size):
                if self.puzzle[i][j] == 0:
                    return i, j
        return None

    def heuristic(self):
        empty_cell = self.find_empty_cell()
        if not empty_cell:
            return None
        return empty_cell

    def solve(self):
        empty_cell = self.heuristic()

        if not empty_cell:
            return True

        row, col = empty_cell

        for num in range(1, self.size + 1):
            if self.is_valid(row, col, num):
                self.puzzle[row][col] = num

                if self.solve():
                    return True

                self.puzzle[row][col] = 0

        return False

    def print_solution(self):
        for row in self.puzzle:
            print(row)

# Example usage:
puzzle = [
    [5, 3, 0, 0, 7, 0, 0, 0, 0],
    [6, 0, 0, 1, 9, 5, 0, 0, 0],
    [0, 9, 8, 0, 0, 0, 0, 6, 0],
    [8, 0, 0, 0, 6, 0, 0, 0, 3],
    [4, 0, 0, 8, 0, 3, 0, 0, 1],
    [7, 0, 0, 0, 2, 0, 0, 0, 6],
    [0, 6, 0, 0, 0, 0, 2, 8, 0],
    [0, 0, 0, 4, 1, 9, 0, 0, 5],
    [0, 0, 0, 0, 8, 0, 0, 7, 9]
]

solver = SudokuSolver(puzzle)
if solver.solve():
    solver.print_solution()
else:
    print("No solution exists.")

```

This program defines a `SudokuSolver` class with methods to check the validity of a number placement, find an empty cell using a heuristic (simplest implementation chooses the first empty cell), solve the puzzle recursively using backtracking, and print the solved puzzle. The example usage demonstrates solving a Sudoku puzzle using the implemented solver.

### Anonymous

Develop a python program to simulate the agent with suitable environment to decide the numbers of papers to be purchased by observing the stock history and prize.



Sure! Here's a Python program to simulate an agent making decisions on the number of papers to purchase based on the observed stock history and price:

python

```
import random

class PaperAgent:
    def __init__(self, initial_cash, initial_stock):
        self.cash = initial_cash
        self.stock = initial_stock
        self.stock_history = [initial_stock]
        self.price_history = []

    def observe_stock_history(self, stock):
        self.stock_history.append(stock)

    def observe_price_history(self, price):
        self.price_history.append(price)

    def decide_purchase(self):
        # Simple decision-making strategy: buy if the current stock is less than the
        previous one
        if len(self.stock_history) >= 2 and self.stock_history[-1] < self.stock_history[-2]:
            # Determine the number of papers to purchase based on available cash and price
            history
            max_purchase = min(self.cash // self.price_history[-1], 10) # Maximum 10 papers
            can be purchased
            if max_purchase > 0:
                purchase_amount = random.randint(1, max_purchase)
                self.stock += purchase_amount
                self.cash -= purchase_amount * self.price_history[-1]
                print(f"Bought {purchase_amount} papers at {self.price_history[-1]} each.")

    def print_status(self):
        print(f"Current cash: {self.cash}")
        print(f"Current stock: {self.stock}")

# Example simulation:
agent = PaperAgent(initial_cash=1000, initial_stock=0)

# Simulate 10 days of stock and price observations
for day in range(1, 11):
    stock = random.randint(5, 20) # Simulate stock availability
    price = random.uniform(1.0, 2.5) # Simulate price fluctuation
    agent.observe_stock_history(stock)
    agent.observe_price_history(price)
    print(f"Day {day}: Stock available: {stock}, Price: {price}")
    agent.decide_purchase()
    agent.print_status()
    print()
```

In this program, the `PaperAgent` class simulates an agent making decisions based on observed stock history and price. The agent starts with an initial amount of cash and stock. It observes the stock and price history each day and decides whether to purchase papers based on a simple strategy (buy if the current stock is less than the previous one). The maximum number of papers that can be purchased is limited to 10 per day. The simulation runs for 10 days, printing the stock availability, price, agent's purchase decision, and current status (cash and stock) for each day.

## Anonymous

Write a python program to represent a priority queue environment and agent to support the Multiple Path Pruning

## ChatGPT

Sure! Here's a Python program to represent a priority queue environment and agent that supports Multiple Path Pruning:

python

```
import heapq

class PriorityQueue:
    def __init__(self):
        self.elements = []

    def empty(self):
        return len(self.elements) == 0

    def put(self, item, priority):
        heapq.heappush(self.elements, (priority, item))

    def get(self):
        return heapq.heappop(self.elements)[1]

class Agent:
    def __init__(self):
        self.visited = set()

    def multiple_path_pruning(self, state):
        # Check if the state has been visited before
        if state in self.visited:
            return True
        # If not visited, add to visited set and return False
        self.visited.add(state)
        return False

# Example usage:
priority_queue = PriorityQueue()
agent = Agent()

# Adding some states to the priority queue
priority_queue.put("State 1", 5)
priority_queue.put("State 2", 3)
priority_queue.put("State 3", 7)
priority_queue.put("State 4", 1)

# Retrieving states from the priority queue
while not priority_queue.empty():
    state = priority_queue.get()
    if not agent.multiple_path_pruning(state):
        print("Processing state:", state)
```

In this program, the `PriorityQueue` class represents the priority queue environment, while the `Agent` class supports Multiple Path Pruning by keeping track of visited states. The `PriorityQueue` uses a heap-based implementation to efficiently retrieve the element with the highest priority. The `Agent` class implements the Multiple Path Pruning strategy by checking if a state has been visited before; if so, it returns `True`, indicating that the state should be pruned. Otherwise, it adds the state to the set of visited states and returns `False`, indicating that the state should be processed. Finally, the example usage demonstrates adding states to the priority queue and retrieving them while applying Multiple Path Pruning.

## Anonymous

Develop a python program to simulate the hierarchical controller environment with the agent to plan and move to the right location

## ChatGPT

Certainly! Below is a Python program simulating a hierarchical controller environment with an agent capable of planning and moving to the right location:

python

```
class HierarchicalController:
    def __init__(self):
        self.locations = {"A": (0, 0), "B": (2, 1), "C": (3, 4)} # Example locations
        self.current_location = None

    def move_to_location(self, location):
        if location in self.locations:
            self.current_location = location
            print(f"Moved to location {location}.")
        else:
            print(f"Location {location} does not exist.")

class Agent:
    def __init__(self, controller):
        self.controller = controller

    def plan_and_move(self, location):
        if self.controller.current_location == location:
            print("Already at the desired location.")
            return

        if location not in self.controller.locations:
            print("Desired location does not exist.")
            return

        # Execute hierarchical planning and move
        print(f"Planning and moving to location {location}.")
        self.controller.move_to_location(location)

# Example usage:
controller = HierarchicalController()
agent = Agent(controller)

# Agent plans and moves to location B
agent.plan_and_move("B")

# Agent plans and moves to location C
agent.plan_and_move("C")

# Trying to move to a non-existing location
agent.plan_and_move("D")
```

This program simulates a hierarchical controller environment with locations A, B, and C. The `HierarchicalController` class manages the locations and moving the agent to the desired location. The `Agent` class interacts with the controller to plan and move to the right location. The `plan_and_move` method of the agent first checks if the desired location exists and then executes hierarchical planning to move to that location if it exists. Finally, an example usage demonstrates the agent planning and moving to locations B and C, and attempting to move to a non-existing location D.

## Anonymous

Design a Python application to model an autonomous agent capable of determining the optimal quantity of raw materials to procure based on historical stock data and market prices.

## ChatGPT

Sure! Here's a Python application to model an autonomous agent capable of determining the optimal quantity of raw materials to procure based on historical stock data and market prices:

python

```
class RawMaterialAgent:
    def __init__(self, initial_cash, initial_stock):
        self.cash = initial_cash
        self.stock = initial_stock
        self.stock_history = [initial_stock]
        self.price_history = []

    def observe_stock_history(self, stock):
        self.stock_history.append(stock)

    def observe_price_history(self, price):
        self.price_history.append(price)

    def determine_optimal_quantity(self):
        # Placeholder for optimization algorithm
        # In a real application, you would use statistical methods or machine learning to
        determine the optimal quantity
        return 10 # For demonstration purposes, let's say the agent decides to buy 10 units
        every time

    def procure_raw_material(self):
        quantity = self.determine_optimal_quantity()
        total_cost = quantity * self.price_history[-1]
        if total_cost <= self.cash:
            self.stock += quantity
            self.cash -= total_cost
            print(f"Procured {quantity} units of raw material at {self.price_history[-1]}
each. Total cost: {total_cost}")
        else:
            print("Insufficient funds to procure raw material.")

    def print_status(self):
        print(f"Current cash: {self.cash}")
        print(f"Current stock: {self.stock}")

# Example usage:
agent = RawMaterialAgent(initial_cash=1000, initial_stock=0)

# Simulate 10 days of stock and price observations
for day in range(1, 11):
    stock = 50 # Simulate constant stock availability for demonstration
    price = round(random.uniform(1.0, 2.5), 2) # Simulate price fluctuation
    agent.observe_stock_history(stock)
    agent.observe_price_history(price)
    print(f"Day {day}: Stock available: {stock}, Price: {price}")
    agent.procure_raw_material()
    agent.print_status()
    print()
```

In this application, the `RawMaterialAgent` class models the autonomous agent capable of determining the optimal quantity of raw materials to procure based on historical stock data and

market prices. The agent starts with an initial amount of cash and stock and observes stock and price history each day. The `determine_optimal_quantity` method is a placeholder for the optimization algorithm that would determine the optimal quantity based on historical data. In this simplified example, the agent always decides to buy 10 units of raw material. The `procure_raw_material` method executes the procurement of raw material based on the determined quantity and available funds. Finally, the example usage simulates 10 days of observations and procurement actions.

## Anonymous

Write a python program to simulate the crossword puzzle problem with 10 words that satisfy suitable constraints in a domain

## ChatGPT

Sure, here's a Python program to simulate a crossword puzzle problem with 10 words that satisfy suitable constraints:

python

```
import random

class CrosswordPuzzle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.grid = [[' ' for _ in range(width)] for _ in range(height)]
        self.words = []

    def add_word(self, word, direction, row, col):
        word_len = len(word)
        if direction == 'across':
            for i in range(word_len):
                self.grid[row][col + i] = word[i]
        elif direction == 'down':
            for i in range(word_len):
                self.grid[row + i][col] = word[i]
        else:
            raise ValueError("Invalid direction. Use 'across' or 'down'.")
        self.words.append((word, direction, row, col))

    def print_grid(self):
        for row in self.grid:
            print(' '.join(row))

# Generate random words
words = ['python', 'program', 'crossword', 'puzzle', 'algorithm', 'simulation',
        'constraint', 'solution', 'problem', 'domain']

# Create a crossword puzzle object
puzzle = CrosswordPuzzle(10, 10)

# Randomly add words to the crossword puzzle
for word in words:
    direction = random.choice(['across', 'down'])
    if direction == 'across':
        row = random.randint(0, 9)
        col = random.randint(0, 10 - len(word))
    else:
        row = random.randint(0, 10 - len(word))
        col = random.randint(0, 9)
```



```
puzzle.add_word(word, direction, row, col)

# Print the crossword puzzle grid
puzzle.print_grid()
```

In this program, the `CrosswordPuzzle` class represents a crossword puzzle with a given width and height. Words are added to the puzzle with specified direction, row, and column. The `print_grid` method prints the puzzle grid. We generate a list of random words and create a `CrosswordPuzzle` object. We then randomly add words to the puzzle and print the resulting grid. This program provides a simple simulation of a crossword puzzle with randomly placed words.

## Anonymous

Write a program to construct a Bayesian Network from a pandas Data Frame containing observed states of variables to model probabilistic dependencies between variables.

## ChatGPT

Sure! Here's a Python program to construct a Bayesian Network from a pandas DataFrame containing observed states of variables:

python

```
import pandas as pd
from pgmpy.models import BayesianModel
from pgmpy.estimators import MaximumLikelihoodEstimator

# Sample DataFrame with observed states of variables
data = {
    'A': [0, 1, 0, 1, 0],
    'B': [1, 1, 0, 0, 1],
    'C': [0, 1, 1, 0, 1],
    'D': [0, 1, 0, 1, 1],
}

df = pd.DataFrame(data)

# Construct a Bayesian Network
model = BayesianModel()

# Estimate the structure of the network using Maximum Likelihood Estimation
estimator = MaximumLikelihoodEstimator(model, df)

# Fit the model to the observed data
model.fit(df, estimator=estimator)

# Print the model structure
print("Bayesian Network Structure:")
for edge in model.edges():
    print(edge)
```

This program constructs a Bayesian Network using the `pgmpy` library, which is a Python library for Probabilistic Graphical Models. We create a sample DataFrame `df` containing observed states of variables A, B, C, and D. We then construct an empty Bayesian Network model and estimate the structure of the network using Maximum Likelihood Estimation. Finally, we fit the model to the observed data and print the structure of the Bayesian Network.

You would need to install the `pgmpy` library if you haven't already. You can install it via pip:

```
pip install pgmpy
```

## Anonymous

Write a program to infer from the Bayesian Network to make probabilistic predictions or queries about unobserved variables given observed evidence

## ChatGPT

Sure! Here's a Python program to infer from the Bayesian Network and make probabilistic predictions or queries about unobserved variables given observed evidence:

```
python
```

```
import pandas as pd
from pgmpy.models import BayesianModel
from pgmpy.inference import VariableElimination

# Sample DataFrame with observed states of variables
data = {
    'A': [0, 1, 0, 1, 0],
    'B': [1, 1, 0, 0, 1],
    'C': [0, 1, 1, 0, 1],
    'D': [0, 1, 0, 1, 1],
}

df = pd.DataFrame(data)

# Construct a Bayesian Network
model = BayesianModel([('A', 'B'), ('C', 'B'), ('C', 'D')])

# Fit the model to the observed data
model.fit(df)

# Perform inference
inference = VariableElimination(model)

# Define the evidence (observed states)
evidence = {'A': 0, 'C': 1}

# Make probabilistic queries about unobserved variables
query_result = inference.query(variables=['B', 'D'], evidence=evidence)
print("Probabilistic predictions:")
print(query_result)
```

This program uses the `pgmpy` library to perform inference from the Bayesian Network and make probabilistic predictions or queries about unobserved variables given observed evidence. We first construct a Bayesian Network model with the known structure ('A' -> 'B', 'C' -> 'B', 'C' -> 'D'). Then, we fit the model to the observed data and perform inference using the Variable Elimination algorithm. We define evidence (observed states) for variables 'A' and 'C' and query for the probabilities of variables 'B' and 'D' given the evidence. Finally, we print the probabilistic predictions.

## Anonymous

Develop a python program to simulate the crossword puzzle problem by assuming the static ordering of nodes that satisfy suitable constraints in a domain and apply any heuristic search

strategy.

## ChatGPT

Sure! Below is a Python program that simulates the crossword puzzle problem by assuming the static ordering of nodes and applying the A\* algorithm as a heuristic search strategy:

python

```
import heapq

class CrosswordPuzzleSolver:
    def __init__(self, puzzle):
        self.puzzle = puzzle
        self.width = len(puzzle[0])
        self.height = len(puzzle)
        self.word_list = []

    def solve(self):
        self.generate_word_list()
        self.word_list.sort(key=lambda x: len(x), reverse=True) # Order words by length
        descending
        self.word_list.sort(key=lambda x: sum(1 for c in x if c == '.')) # Order words by
        number of blank spaces ascending

        return self.astar_search()

    def generate_word_list(self):
        for row in self.puzzle:
            word = ''
            for char in row:
                if char == '.':
                    word += '.'
                elif len(word) > 1:
                    self.word_list.append(word)
                    word = ''
                else:
                    word = ''
            if len(word) > 1:
                self.word_list.append(word)

        for col in range(self.width):
            word = ''
            for row in range(self.height):
                if self.puzzle[row][col] == '.':
                    word += '.'
                elif len(word) > 1:
                    self.word_list.append(word)
                    word = ''
                else:
                    word = ''
            if len(word) > 1:
                self.word_list.append(word)

    def heuristic(self, state):
        # Simple heuristic: prioritize states with fewer blank spaces remaining
        return sum(1 for row in state for char in row if char == '.')

    def neighbors(self, state):
        neighbors = []
        for word in self.word_list:
            if len(word) <= self.width:
                for row in range(self.height):
                    for col in range(self.width - len(word) + 1):
                        new_state = [list(row_state) for row_state in state]
                        valid = True
                        for i, char in enumerate(word):
                            if new_state[row][col + i] != '.' and new_state[row][col + i] != char:
                                valid = False
                                break
                        if valid:
                            new_state[row][col:col + len(word)] = list(word)
                            neighbors.append(new_state)

        char:
```

```

        valid = False
        break
        new_state[row][col + i] = char
    if valid:
        neighbors.append((new_state, word))

    return neighbors

def is_goal(self, state):
    return all('.' not in row for row in state)

def reconstruct_path(self, came_from, current_state):
    path = []
    while current_state in came_from:
        path.append(current_state)
        current_state = came_from[current_state]
    return path[::-1]

def astar_search(self):
    start_state = [['.' for _ in range(self.width)] for _ in range(self.height)]
    open_set = [(self.heuristic(start_state), start_state)]
    heapq.heapify(open_set)
    came_from = {}
    g_scores = {start_state: 0}

    while open_set:
        _, current_state = heapq.heappop(open_set)
        if self.is_goal(current_state):
            return self.reconstruct_path(came_from, current_state)
        for neighbor_state, word in self.neighbors(current_state):
            tentative_g = g_scores[current_state] + 1 # Assuming uniform cost for
            if neighbor_state not in g_scores or tentative_g < g_scores[neighbor_state]:
                came_from[neighbor_state] = current_state
                g_scores[neighbor_state] = tentative_g
                f_score = tentative_g + self.heuristic(neighbor_state)
                heapq.heappush(open_set, (f_score, neighbor_state))

    return None

# Example usage:
puzzle = [
    ['.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.'],
    ['.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.'],
    ['.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.'],
    ['.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.'],
    ['.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.'],
    ['.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.'],
    ['.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.'],
    ['.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.'],
    ['.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.'],
    ['.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.'],
]

solver = CrosswordPuzzleSolver(puzzle)
solution = solver.solve()
if solution:
    print("Solution found:")
    for row in solution:
        print(row)
else:
    print("No solution found.")

```

This program simulates the crossword puzzle problem by assuming the static ordering of nodes and applying the A\* algorithm as a heuristic search strategy. The `CrosswordPuzzleSolver` class represents the solver, which uses the A\* algorithm to find the optimal solution. The heuristic used prioritizes states with fewer blank spaces remaining. The example usage demonstrates solving a crossword puzzle with a given grid.

## Anonymous

Write a python program to simulate the crossword puzzle problem that satisfy suitable constraints with any stochastic search strategy and conflict resolution.

## ChatGPT

Sure! Below is a Python program that simulates the crossword puzzle problem and uses simulated annealing as the stochastic search strategy with conflict resolution:

python

```
import random
import math

class CrosswordPuzzleSolver:
    def __init__(self, puzzle):
        self.puzzle = puzzle
        self.width = len(puzzle[0])
        self.height = len(puzzle)
        self.word_list = []
        self.temperature = 1.0
        self.cooling_rate = 0.999

    def solve(self):
        self.generate_word_list()
        self.word_list.sort(key=lambda x: len(x), reverse=True) # Order words by length
        descending
        self.word_list.sort(key=lambda x: sum(1 for c in x if c == '.')) # Order words by
        number of blank spaces ascending

        current_state = self.initialize_state()
        best_state = current_state

        while self.temperature > 0.1:
            neighbor_state = self.get_neighbor(current_state)
            current_score = self.heuristic(current_state)
            neighbor_score = self.heuristic(neighbor_state)
            if neighbor_score >= current_score:
                current_state = neighbor_state
            else:
                if random.random() < math.exp((neighbor_score - current_score) /
self.temperature):
                    current_state = neighbor_state
                if self.heuristic(current_state) > self.heuristic(best_state):
                    best_state = current_state
                self.temperature *= self.cooling_rate

        return best_state

    def initialize_state(self):
        return [['.' for _ in range(self.width)] for _ in range(self.height)]

    def generate_word_list(self):
        for row in self.puzzle:
            word = ''
            for char in row:
                if char == '.':
                    word += '.'
                elif len(word) > 1:
                    self.word_list.append(word)
                    word = ''
                else:
                    word = ''
            if len(word) > 1:
                self.word_list.append(word)

        for col in range(self.width):
            word = ''
```

```

        for row in range(self.height):
            if self.puzzle[row][col] == '.':
                word += '.'
            elif len(word) > 1:
                self.word_list.append(word)
                word = ''
            else:
                word = ''
        if len(word) > 1:
            self.word_list.append(word)

    def heuristic(self, state):
        # Simple heuristic: prioritize states with fewer blank spaces remaining
        return sum(1 for row in state for char in row if char == '.')

    def get_neighbor(self, state):
        neighbor_state = [list(row) for row in state]
        word = random.choice(self.word_list)
        while True:
            row = random.randint(0, self.height - 1)
            col = random.randint(0, self.width - 1)
            if len(word) <= self.width - col:
                valid = True
                for i, char in enumerate(word):
                    if neighbor_state[row][col + i] != '.' and neighbor_state[row][col + i]
!= char:
                        valid = False
                        break
                if valid:
                    for i, char in enumerate(word):
                        neighbor_state[row][col + i] = char
                    break
        return neighbor_state

# Example usage:
puzzle = [
    [ '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', ],
    [ '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', ],
    [ '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', ],
    [ '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', ],
    [ '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', ],
    [ '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', ],
    [ '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', ],
    [ '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', ],
    [ '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', ],
    [ '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', '.', ],
]

solver = CrosswordPuzzleSolver(puzzle)
solution = solver.solve()
if solution:
    print("Solution found:")
    for row in solution:
        print(row)
else:
    print("No solution found.")

```

This program simulates the crossword puzzle problem using simulated annealing as the stochastic search strategy with conflict resolution. The `CrosswordPuzzleSolver` class represents the solver, which iteratively tries to find a better state using simulated annealing until a termination condition is met. The temperature parameter controls the probability of accepting a worse solution, and the cooling rate controls the rate at which the temperature decreases. The example usage demonstrates solving a crossword puzzle with a given grid.

Develop a command-line task management system that allows users to add, view, mark tasks as completed, and receive notifications for upcoming deadlines.

## ChatGPT

Sure! Below is a basic command-line task management system implemented in Python:

python

```
import datetime

class Task:
    def __init__(self, description, deadline):
        self.description = description
        self.deadline = deadline
        self.completed = False

    def mark_as_completed(self):
        self.completed = True

    def is_upcoming(self):
        return datetime.datetime.now() < self.deadline

class TaskManager:
    def __init__(self):
        self.tasks = []

    def add_task(self, task):
        self.tasks.append(task)

    def view_tasks(self):
        if not self.tasks:
            print("No tasks.")
        else:
            for index, task in enumerate(self.tasks):
                print(f"{index + 1}. {task.description} - Deadline: {task.deadline.strftime('%Y-%m-%d')} - {'Completed' if task.completed else 'Not Completed'}")

    def mark_task_completed(self, index):
        if 1 <= index <= len(self.tasks):
            self.tasks[index - 1].mark_as_completed()
            print("Task marked as completed.")
        else:
            print("Invalid task index.")

    def upcoming_deadlines(self):
        upcoming_tasks = [task for task in self.tasks if task.is_upcoming()]
        if upcoming_tasks:
            print("Upcoming deadlines:")
            for task in upcoming_tasks:
                print(f"{task.description} - Deadline: {task.deadline.strftime('%Y-%m-%d')}")
        else:
            print("No upcoming deadlines.")

def main():
    task_manager = TaskManager()

    while True:
        print("\nTask Management System")
        print("1. Add Task")
        print("2. View Tasks")
        print("3. Mark Task as Completed")
        print("4. View Upcoming Deadlines")
        print("5. Exit")

        choice = input("Enter your choice: ")

        if choice == '1':
```

```

        description = input("Enter task description: ")
        deadline_str = input("Enter deadline (YYYY-MM-DD): ")
        deadline = datetime.datetime.strptime(deadline_str, '%Y-%m-%d')
        task = Task(description, deadline)
        task_manager.add_task(task)
        print("Task added successfully.")
    elif choice == '2':
        task_manager.view_tasks()
    elif choice == '3':
        index = int(input("Enter task index to mark as completed: "))
        task_manager.mark_task_completed(index)
    elif choice == '4':
        task_manager.upcoming_deadlines()
    elif choice == '5':
        print("Exiting...")
        break
    else:
        print("Invalid choice. Please try again.")

if __name__ == "__main__":
    main()

```

This program allows users to perform various task management operations via the command line. Users can add tasks, view existing tasks, mark tasks as completed, view upcoming deadlines, and exit the program. Tasks are represented by the `Task` class, which includes a description, deadline, and completion status. The `TaskManager` class provides methods to manage tasks, including adding, viewing, marking as completed, and listing upcoming deadlines. The `main` function serves as the entry point to the program, displaying a menu of options and handling user input.

### Anonymous

Develop a python program to simulate a knowledge base with a list of clauses in order to make top-down inference, also creates a dictionary that maps each atom into the set of clauses with that atom in the head.

### ChatGPT

Sure! Below is a Python program that simulates a knowledge base with a list of clauses and creates a dictionary that maps each atom into the set of clauses with that atom in the head:

python

```

class Clause:
    def __init__(self, head, body=[]):
        self.head = head
        self.body = body

class KnowledgeBase:
    def __init__(self):
        self.clauses = []
        self.atom_to_clauses = {}

    def add_clause(self, clause):
        self.clauses.append(clause)
        if clause.head not in self.atom_to_clauses:
            self.atom_to_clauses[clause.head] = set()
        self.atom_to_clauses[clause.head].add(clause)

    def query(self, atom):
        if atom in self.atom_to_clauses:
            return self.atom_to_clauses[atom]

```



```

else:
    return set()

# Example usage:
kb = KnowledgeBase()

# Adding clauses to the knowledge base
kb.add_clause(Clause('p', ['q', 'r']))
kb.add_clause(Clause('q', ['s']))
kb.add_clause(Clause('r', ['t']))
kb.add_clause(Clause('s'))
kb.add_clause(Clause('t'))

# Querying the knowledge base for clauses with 'p' in the head
p_clauses = kb.query('p')
print("Clauses with 'p' in the head:")
for clause in p_clauses:
    print(f"{clause.head} :- {' '.join(clause.body)}")

```

In this program, the `Clause` class represents a logical clause with a head and an optional body. The `KnowledgeBase` class maintains a list of clauses and creates a dictionary (`atom_to_clauses`) that maps each atom into the set of clauses with that atom in the head. The `add_clause` method adds a new clause to the knowledge base and updates the dictionary accordingly. The `query` method allows querying the knowledge base for clauses with a specific atom in the head.

In the example usage, we create a knowledge base (`kb`) and add several clauses to it. Then, we query the knowledge base for clauses with 'p' in the head and print the results.



### Anonymous

Write a python program to demonstrate the belief network for making an uncertain decision.



### ChatGPT

Certainly! Below is a Python program demonstrating a simple belief network for making an uncertain decision:

python

```

from pgmpy.models import BayesianModel
from pgmpy.factors.discrete import TabularCPD
from pgmpy.inference import VariableElimination

# Define the Bayesian Network structure
model = BayesianModel([('Alarm', 'Fire'), ('Alarm', 'Burglary')])

# Define the Conditional Probability Distributions (CPDs)
cpd_alarm = TabularCPD(variable='Alarm', variable_card=2, values=[[0.95], [0.05]])
cpd_fire = TabularCPD(variable='Fire', variable_card=2, values=[[0.9, 0.7], [0.1, 0.3]],
evidence=['Alarm'], evidence_card=[2])
cpd_burglary = TabularCPD(variable='Burglary', variable_card=2, values=[[0.8, 0.1], [0.2, 0.9]],
evidence=['Alarm'], evidence_card=[2])

# Add CPDs to the model
model.add_cpds(cpd_alarm, cpd_fire, cpd_burglary)

# Check model for consistency
assert model.check_model()

# Perform Variable Elimination for inference
inference = VariableElimination(model)

```

```
# Query for the probability of a fire given the alarm has gone off
query_result = inference.query(variables=['Fire'], evidence={'Alarm': 1})
print("Probability of fire given the alarm has gone off:")
print(query_result)
```

In this program, we define a belief network structure representing the relationships between the variables 'Alarm', 'Fire', and 'Burglary'. We define the Conditional Probability Distributions (CPDs) for each variable and add them to the model. The CPDs represent the probabilities of different outcomes given certain evidence. We then perform Variable Elimination for inference, which allows us to query the model to determine the probability of a fire given that the alarm has gone off.

## Anonymous

Develop a command-line personal journal and mood tracker that enables users to record daily entries, track their mood, and visualize mood trends.

## ChatGPT

Below is a basic command-line personal journal and mood tracker implemented in Python:

python

```
import datetime

class Entry:
    def __init__(self, date, content, mood):
        self.date = date
        self.content = content
        self.mood = mood

class Journal:
    def __init__(self):
        self.entries = []

    def add_entry(self, entry):
        self.entries.append(entry)

    def view_entries(self):
        if not self.entries:
            print("No entries.")
        else:
            for entry in self.entries:
                print(f>Date: {entry.date.strftime('%Y-%m-%d')}")
                print(f>Content: {entry.content}")
                print(f>Mood: {entry.mood}")
                print()

    def track_mood(self):
        mood = input("Enter your mood for today: ")
        return mood

    def visualize_mood_trends(self):
        mood_counts = {}
        for entry in self.entries:
            if entry.mood in mood_counts:
                mood_counts[entry.mood] += 1
            else:
                mood_counts[entry.mood] = 1

        print("Mood Trends:")
        for mood, count in mood_counts.items():
            print(f">{mood}: {count}")
```

```
def main():
    journal = Journal()

    while True:
        print("\nPersonal Journal and Mood Tracker")
        print("1. Add Entry")
        print("2. View Entries")
        print("3. Track Mood")
        print("4. Visualize Mood Trends")
        print("5. Exit")

        choice = input("Enter your choice: ")

        if choice == '1':
            content = input("Enter journal entry content: ")
            mood = journal.track_mood()
            entry = Entry(datetime.datetime.now(), content, mood)
            journal.add_entry(entry)
            print("Entry added successfully.")
        elif choice == '2':
            journal.view_entries()
        elif choice == '3':
            mood = journal.track_mood()
            print(f"Mood tracked: {mood}")
        elif choice == '4':
            journal.visualize_mood_trends()
        elif choice == '5':
            print("Exiting...")
            break
        else:
            print("Invalid choice. Please try again.")

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

This program allows users to record daily journal entries, track their mood, and visualize mood trends via the command line. The `Entry` class represents a journal entry with attributes for date, content, and mood. The `Journal` class maintains a list of entries and provides methods to add entries, view entries, track mood, and visualize mood trends. The `main` function serves as the entry point to the program, displaying a menu of options and handling user input.

### Anonymous

Design a Python program to implement uninformed search algorithms such as Depth- First Search (DFS)

### ChatGPT

Sure, here's a Python program implementing Depth-First Search (DFS) algorithm for an undirected graph:

python

```
class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v):
        if u not in self.graph:
            self.graph[u] = []
        if v not in self.graph:
            self.graph[v] = []
```

```

        self.graph[u].append(v)
        self.graph[v].append(u)

    def dfs(self, start):
        visited = set()
        stack = [start]
        traversal = []

        while stack:
            vertex = stack.pop()
            if vertex not in visited:
                visited.add(vertex)
                traversal.append(vertex)
                stack.extend(reversed(self.graph[vertex]))

        return traversal

# Example usage:
g = Graph()
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 3)
g.add_edge(1, 4)
g.add_edge(2, 4)
g.add_edge(3, 4)

print("Depth-First Search Traversal:")
print(g.dfs(0))

```

In this program, the `Graph` class represents an undirected graph. The `add_edge` method is used to add edges between vertices. The `dfs` method implements the Depth-First Search algorithm starting from a specified vertex. It maintains a stack to keep track of vertices to visit and a set to keep track of visited vertices. The traversal list keeps track of the order in which vertices are visited. Finally, we create a graph, add some edges, and perform DFS starting from vertex 0.



### Anonymous

Design a Python program to implement uninformed search algorithms such as Breadth- First Search (BFS)



### ChatGPT

Sure, here's a Python program implementing Breadth-First Search (BFS) algorithm for an undirected graph:

python

```

from collections import deque

class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v):
        if u not in self.graph:
            self.graph[u] = []
        if v not in self.graph:
            self.graph[v] = []
        self.graph[u].append(v)
        self.graph[v].append(u)

    def bfs(self, start):

```

```

visited = set()
queue = deque([start])
traversal = []

while queue:
    vertex = queue.popleft()
    if vertex not in visited:
        visited.add(vertex)
        traversal.append(vertex)
        queue.extend(self.graph[vertex])

return traversal

# Example usage:
g = Graph()
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 3)
g.add_edge(1, 4)
g.add_edge(2, 4)
g.add_edge(3, 4)

print("Breadth-First Search Traversal:")
print(g.bfs(0))

```

In this program, the `Graph` class represents an undirected graph. The `add_edge` method is used to add edges between vertices. The `bfs` method implements the Breadth-First Search algorithm starting from a specified vertex. It maintains a queue to keep track of vertices to visit and a set to keep track of visited vertices. The traversal list keeps track of the order in which vertices are visited. Finally, we create a graph, add some edges, and perform BFS starting from vertex 0.

### Anonymous

Design a python program to implement the A\* algorithm, an informed search algorithm used for pathfinding

### ChatGPT

Certainly! Here's a Python program implementing the A\* algorithm for pathfinding:

python

```

import heapq

class GridNode:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.g = float('inf') # Cost from start node to current node
        self.h = 0 # Heuristic cost from current node to goal node
        self.f = float('inf') # Estimated total cost from start to goal through current
node
        self.parent = None # Parent node

    def __lt__(self, other):
        return self.f < other.f

class AStar:
    def __init__(self, grid, start, goal):
        self.grid = grid
        self.start = start
        self.goal = goal
        self.open_set = [] # Priority queue for nodes to be evaluated

```

```

self.closed_set = set() # Set of nodes already evaluated

def heuristic(self, node):
    # Manhattan distance heuristic
    return abs(node.x - self.goal.x) + abs(node.y - self.goal.y)

def get_neighbors(self, node):
    neighbors = []
    for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
        x, y = node.x + dx, node.y + dy
        if 0 <= x < len(self.grid) and 0 <= y < len(self.grid[0]) and self.grid[x][y] !=
1:
            neighbors.append(GridNode(x, y))
    return neighbors

def reconstruct_path(self, current):
    path = []
    while current is not None:
        path.append((current.x, current.y))
        current = current.parent
    return path[::-1]

def search(self):
    start_node = GridNode(self.start[0], self.start[1])
    start_node.g = 0
    start_node.h = self.heuristic(start_node)
    start_node.f = start_node.g + start_node.h

    heapq.heappush(self.open_set, start_node)

    while self.open_set:
        current = heapq.heappop(self.open_set)

        if (current.x, current.y) == self.goal:
            return self.reconstruct_path(current)

        self.closed_set.add((current.x, current.y))

        for neighbor in self.get_neighbors(current):
            if (neighbor.x, neighbor.y) in self.closed_set:
                continue

            tentative_g = current.g + 1 # Cost of moving to neighbor is always 1

            if tentative_g < neighbor.g:
                neighbor.parent = current
                neighbor.g = tentative_g
                neighbor.h = self.heuristic(neighbor)
                neighbor.f = neighbor.g + neighbor.h

                if neighbor not in self.open_set:
                    heapq.heappush(self.open_set, neighbor)

    return None

# Example usage:
grid = [
    [0, 0, 0, 0, 0],
    [0, 1, 1, 0, 0],
    [0, 0, 0, 0, 0],
    [0, 0, 1, 1, 1],
    [0, 0, 0, 0, 0]
]

start = (0, 0)
goal = (4, 4)

astar = AStar(grid, start, goal)
path = astar.search()

if path:
    print("Path found:")
    for node in path:
        print(node)

```

```
else:  
    print("No path found.")
```

In this program, the `GridNode` class represents a node in the grid. The `AStar` class implements the A\* algorithm for pathfinding on a grid. It maintains an open set (priority queue) of nodes to be evaluated and a closed set of nodes already evaluated. The `heuristic` method calculates the Manhattan distance heuristic from a node to the goal node. The `get_neighbors` method retrieves neighboring nodes of a given node. The `search` method performs the A\* search to find the optimal path from the start node to the goal node. Finally, we create a grid, start and goal positions, and use the A\* algorithm to find the optimal path.

[| Terms of use](#) | [Privacy policy](#) |