Search strategies implementation in Python and logic programming in Prolog

Part I: The status of chess as a topic in artificial intelligence

1. Deep Blue

A milestone in the research of chess was made by Feng-Hsiung Hsu and his team back in 1997 when Deep Blue, their specialized computer, beat the then world champion Garry Kasparov. In order to achieve this, important study was done on graph expansion and decision making techniques. Both related to the area of search. The highlights include, but are not limited to, the following:

- When the machine beat Garry Kasparov, it searched about 200 million chess positions per second, using 480 custom chess chips, with each chip searching up to 3 million positions per second.
- Ad hoc handwritten techniques for selective pruning were surpassed in performance by simple-minded brute force search subject to normal alpha-beta pruning.
- Selective extensions emerged as an alternative to selective prunning. Which spent *less time searching the unpromising moves* by just searching "interesting" lines deeper *without completely discarding them*.
- Feng-Hsiung Hsu introduced the idea of *singular extensions* in 1986 which used *test searches* to perform searches not necessarily needed for regular search but useful to measure the forcefulness of the moves. Then, for example, if test searches suggest only one single "good" move is possible, then the "good" move deserves to be searched deeper.
- A form of test search is *null move pruning* where the test searches have one player making two moves in a row, with the opponent thus making a null movement in between. If the result is "unsatisfactory", then the first move by the player is unpromising and can be pruned away.
- After chess was solved–i.e. a program was good enough to beat a world champion—harder games like the Japanese Shogi and the Chinese Go were suggested as next targets and 10 years solved by the use of neural networks by Deep Mind's AlphaGo.
- AlphaGo used two deep neural networks: one for evaluation and the other for deciding which move to include in the search.
- Deep Neural Networks serve as a proof that small improvements in a few areas, amplified by greater computation power, create huge differences.

Source: Hsu, F. (2022). Behind Deep Blue: Building the Computer That Defeated the World Chess Champion. Princeton University Press.

2. Why bother improving search? The state-space complexity of chess

Chess is a game of immense complexity, with a vast number of possible game states and moves. The state-space complexity of chess is estimated to be around 10^46.25. This is the number of legal chess positions, a number so large that it's impossible to compute all of them. The game-tree complexity, which is the total number of possible games, is even more astronomical, estimated to be around 10^123. This is more than the number of atoms in the observable universe!

The complexity of chess makes it a perfect testbed for search algorithms. The goal of these algorithms is to navigate through this vast search space and find the best move in any given position. The better the search algorithm, the stronger the chess-playing AI can be. However, the brute force approach, which involves searching through all possible moves, is not feasible due to the game's complexity. Therefore, search strategies in chess AI focus on pruning the search tree and focusing on the most promising moves.

The minimax algorithm, for instance, is a recursive algorithm used for decision making in game theory and artificial intelligence that simulates all possible games to determine the best move, assuming that the opponent is also playing optimally. To improve the efficiency of the minimax algorithm, alpha-beta pruning is often used. This technique eliminates branches in the game tree that do not need to be explored because there already exists a better move available.

Another significant algorithm in chess AI is the Monte Carlo Tree Search (MCTS). This algorithm uses random sampling as part of the decision-making process. Unlike the minimax algorithm, MCTS does not need to evaluate all possible games. Instead, it uses statistical analysis of sample games to determine the best move. This makes MCTS particularly effective in complex game scenarios where the total number of possible games is too large to compute.

Despite the complexity, chess has been effectively "solved" by AI, with programs like Deep Blue and AlphaZero able to beat world champion human players. This success has led to AI researchers moving on to even more complex games like Go and Shogi.

Sources:

MIT

Scientific American

Chess Programming

Core

Wikipedia

Highlights in Science, Engineering and Technology - Tree Search Algorithms For Chinese Chess

3. Chess byproducts: endgame tablebases

Endgame tablebases are a significant byproduct of chess research and have been instrumental in solving complex endgame scenarios. They are essentially databases that contain the exact evaluation of all possible positions with a limited number of pieces on the board. The use of endgame tablebases has revolutionized the way chess endgames are studied and played, providing definitive answers to positions that were once considered too complex to analyze.

The development of 8-piece endgame tablebases has been a significant milestone in this area. These tablebases contain all possible positions with eight pieces on the board, including the kings. The creation of these tablebases was a massive computational task, requiring the analysis of trillions of positions. The results have been fascinating, revealing new theoretical wins and drawing lines in positions that were previously thought to be decided.

One of the most intriguing findings from the 8-piece endgame tablebases is the discovery of positions that require more than 500 moves to convert an advantage into a win. These positions, which are far beyond the 50-move rule applied in practical play, provide fascinating insights into the depth and complexity of chess.

The use of endgame tablebases extends beyond pure theoretical interest. They are used in practical play by top chess engines to play the endgame perfectly. They also serve as a valuable tool for chess players and researchers to study and understand the endgame better.

The creation of endgame tablebases is an ongoing process, with researchers continually working on larger tablebases. As the tablebases grow, they continue to reveal the immense complexity and beauty of chess, providing new insights and challenges for players and researchers.

Sources:

Chessbase - 8-piece endgame tablebases: first findings and interview

Chessbase - Cooks and finds with 8-piece tablebases

Chessbase - Study of the month: Endgame studies, endgame theory

4. Beyond Chess: The Evolution of Game AI

The success of AI in chess has paved the way for its application in other games and even beyond the realm of games. From classic board games like Go and Shogi to modern video games like StarCraft and Dota 2, AI has made significant strides. The complexity and diversity of these games provide new challenges and opportunities for AI research.

In these games, AI has to deal with more complex state spaces, real-time decision-making, and multi-agent environments. For instance, in StarCraft, an AI player needs to manage resources, control multiple units simultaneously, and adapt to the strategies of the opponent. These tasks require advanced AI techniques such as deep reinforcement learning and multi-agent learning.

Moreover, the advances in Game AI are not confined to games. They are being extended to other areas such as robotics and chemical synthesis. For example, the techniques used to solve the Rubik's cube with a robot hand are similar to those used in Game AI.

The evolution of Game AI also highlights the importance of improving search strategies. As games become more complex, the search space becomes larger and more challenging to navigate. Therefore, developing more efficient and effective search algorithms remains a crucial task in AI research.

Source: Springer

Part II: Search strategies implemented in python for the 8-Tile puzzle (snippets from the code)

General Purpose Code

The following functions are kept identical in BFS and DFS implementations. Both are in charge of "moving the tiles", i.e. changing the state of the board. For the A* search, a change is necessary only in the search function, as the apply_action() remains the same.

In [ ]:
```python
# Define the search function
def search(state):
    frontier.put(state)
    G.add_node(state)
    while not frontier.empty():
        state = frontier.get()
        if state == GOAL_STATE:
            return True
        for action in ACTIONS:
            new_state = apply_action(state, action)
            if new_state is not None and new_state not in G:
                G.add_edge(state, new_state)
                frontier.put(new_state)
    return False

# Define the action function
def apply_action(state, action):
    new_state = [list(row) for row in state]
    i, j = next((i, j) for i, row in enumerate(state) for j, cell in enumera
    if action == 'up' and i > 0:
        new_state[i][j], new_state[i - 1][j] = new_state[i - 1][j], new_stat
    elif action == 'down' and i < 2:
        new_state[i][j], new_state[i + 1][j] = new_state[i + 1][j], new_stat
    elif action == 'left' and j > 0:
        new_state[i][j], new_state[i][j - 1] = new_state[i][j - 1], new_stat
    elif action == 'right' and j < 2:
        new_state[i][j], new_state[i][j + 1] = new_state[i][j + 1], new_stat
    else:
        return None
    return tuple(map(tuple, new_state))

ACTIONS = ['up', 'down', 'left', 'right']
```

1. BFS and DFS

The most important consideration resulting from this implementation is to understand that the difference between a BFS and DFS lies on the data structure used to keep track of new_states. For BFS, a queue is used to force the search go through all the nodes by 'width'. On the other hand, DFS uses a stack to go through the nodes by 'height'. It is important to note that a deque is an equivalent to a stack in python. Both part of the *collections* library of python.

In [ ]:
```
# Create a queue for the frontier
frontier = Queue()

# Create a stack for the frontier
frontier = deque()
```

2. A* Search

The task here was to implement three heuristics for the evaluation function. To achieve so, let's first define the search in terms of a cost associated to a candidate solution. Let's also note that the right data structure for this task is the Priority_queue:

In [ ]:
```
#Define the search function for A* search
def search(state, g, heuristic):
    # If the state is the goal state, return the cost
    if state == GOAL_STATE:
        return g

    # Generate the state space
    for action in ACTIONS:
        new_state = apply_action(state, action)
        if new_state is not None and new_state not in G:
            G.add_edge(state, new_state)
            cost = g + 1 + heuristic(new_state)
            frontier.put((cost, new_state, g + 1))
    # Continue the search
    while not frontier.empty():
        _, state, g = frontier.get()
        cost = search(state, g, heuristic)
        if cost is not None:
            return cost
```

As an extra consideration, the heuristic received as a parameter of the search function will be the responsible of determining the priority associated to an element added to the priority queue.

In [ ]:
```
# Create a priority queue for the frontier
    frontier = PriorityQueue()
```

Now, as the cost is calculated in function of a heuristic(), the heuristic can be defined separately to then assign it to the function. The assignation is achieved inside a loop which initializes all important variables (this way avoiding memory mismanagement) and then calling to the locally defined switch function to obtain a heuristic and pass it as an argument to the search.

In [ ]:
```
#...more code

def Manhattan_heuristic(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                x, y = GOAL_POS[state[i][j]]
                distance += abs(x - i) + abs(y - j)
    return distance

def Euclidean_heuristic(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                x, y = GOAL_POS[state[i][j]]
                distance += math.sqrt((x - i)**2 + (y - j)**2)
    return distance

def Max_heuristic(state):
    manhattan_distance = 0
    misplaced_tiles = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                x, y = GOAL_POS[state[i][j]]
                manhattan_distance += abs(x - i) + abs(y - j)
                if (i, j) != (x, y):
                    misplaced_tiles += 1
    return max(manhattan_distance, misplaced_tiles)

def Linear_Conflict_Heuristic(state):
    distance = 0
    row_conflicts = [0]*3
    column_conflicts = [0]*3
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                # Calculate the Manhattan distance
                x, y = GOAL_POS[state[i][j]]
                distance += abs(x - i) + abs(y - j)

                # Check for linear conflicts
                if state[i][j] in goal_row_order[i]:
                    for k in range(j+1, 3):
                        if state[i][k] in goal_row_order[i] and state[i][j]
                            row_conflicts[i] += 1
                if state[j][i] in goal_column_order[i]:
                    for k in range(j+1, 3):
                        if state[k][i] in goal_column_order[i] and state[j][
                            column_conflicts[i] += 1
    linear_conflicts = sum(row_conflicts) + sum(column_conflicts)
    return distance + 2*linear_conflicts

def default():
    print("Invalid choice. Please try again.")

switch = {
    "1": Manhattan_heuristic,
    "2": Euclidean_heuristic,
    "3": Max_heuristic,
    "4": Linear_Conflict_Heuristic,
}

#...more code
```

In [ ]:

```
while True:

    # Create a priority queue for the frontier
    frontier = PriorityQueue()

    # Create a graph to store the state space
    G = nx.Graph()

    print("Please select a heuristic:")
    print("1. Manhattan")
    print("2. Euclidean")
    print("3. Max")
    print("4. Linear Conflict")
    print("5. Exit")

    choice = input("Enter your choice: ")

    if choice == "5":
        print("Exiting...")
        break
    else:
        heuristic_function = switch.get(choice, default)
        cost = search(INITIAL_STATE, 0, heuristic_function)

        #...more code
```

Part III: Recursion and Backtracking for the N-Queens problem in Python (snippets from the code)

The whole solution for the problem can be reduced to a single function. Some of the highlights of this approach are:

- It works from an initial three empty lists where the contrains will be saved. It allows them to be checked everytime a position is tested for a queen.
- The queens list is where the resulting column position will be saved for the queens in the row corresponding to their index in the list.
- xy_sum keeps track of the bottom-left to upper-right diagonal contrains and xy_dif tracks the upper_left to bottom_right diagonal constrains.
- The recursions works with backtracking using list concatenation. i.e. when solve() is called recursively, the arguments it receives create a new list without modifying the original list. This way, whenever solve reaches a dead state, it can go back check the list containing the elements in the previous call and continue from there.

In [ ]:
```
def n_queens(N):
    def solve(queens, xy_dif, xy_sum):
        p = len(queens)
        if p == N:
            result.append(queens)
            return None
        for q in range(N):
            #print("q: {q}, p: {p}, queens: {queens}, xy_dif: {xy_dif}, xy_s
            if q not in queens and p-q not in xy_dif and p+q not in xy_sum:
                solve(queens+[q], xy_dif+[p-q], xy_sum+[p+q])

    result = []
    solve([],[],[])
    return result
```

Part IV: Implementation of Logic problems in prolog (snippets from the code)

1. The Farmer, Wolf, Goat, Cabbage problem

The highlighs can be summarized as follows:

- The code searches the solution using the *path/3* predicate which recibes a state and, *if state == goal*, returns the solution. Otherwise, it keeps looking for states til completion.

- To change between states, different move functions are implemented, one for each possible scenario: farmer takes wolf, farmer takes goat, farmer takes cabbage, farmer takes self. This has been done through the creation of *move/2* predicates which will attempt to get to the goal. If a dead point is reached, then it backtracks to the previous position and tries again. A *not(member_queue/2)* predicate has been implemented as a predicate to avoid checking over a previously checked state.
- The constrains of the problem are defined through the *unsafe/1* and the *opp/2* predicates.
- Note that this code uses the adts.pl file provided as a code resource for George Luger's book.

In [ ]:
```
%...more code

path(Goal,Goal,Been_queue) :-
    write('Solution Path Is:' ), nl,
    print_queue(Been_queue).

path(State,Goal,Been_queue) :-
    move(State,Next_state),
    not(member_queue(Next_state,Been_queue)),
    add_to_queue(Next_state, Been_queue, New_been_queue),
    path(Next_state,Goal,New_been_queue),!.

move(state(X,X,G,C), state(Y,Y,G,C))
              :- opp(X,Y), not(unsafe(state(Y,Y,G,C))),
                 writelist(['try farmer takes wolf',Y,Y,G,C]).

%...more moves

unsafe(state(X,Y,Y,C)) :- opp(X,Y).

%...

opp(e,w).

%...print the solution
```

2. The cannibals and missionaries problem

An important note here is that it was attempted to reach a solution through the sole modification of the F_W_G_C problem though the difficulty of the task forced to start over the implementation. Thereby, the constrains were defined explicitly in a way that only two move predicates were required instead of one for each possible river crossing state. The most important highlights are:

- A *valid/1* predicate checks that both sides of the river have enough missionaries to avoid being eaten by the cannibals.
- The *move/3* predicates work for either moving from coast 0 to 1 or 1 to 0 (thus resulting in only two move possibilities) for which the passengers of the trip are

limited to the list [DM, DC] as part of list containing all possible combinations of missionaries and cannibals.

- the *solve/4* predicate starts the *depthfirst/6* predicate which goes through the nodes checking wheter or not the current node equals the goal.

In [ ]:
```
%...more code

% Valid state
valid(state(M, C, _)) :-
    M >= 0, C >= 0, M =< 3, C =< 3,
    (M >= C ; M = 0),
    M2 is 3 - M, C2 is 3 - C,
    (M2 >= C2 ; M2 = 0).

% Possible moves
move(state(M, C, B), state(M2, C2, B2), Move) :-
    B = 0, B2 is 1, % Move from original side to other side
    member([DM, DC], [[0, 1], [0, 2], [1, 0], [1, 1], [2, 0]]), % Possible c
    M2 is M - DM, C2 is C - DC, valid(state(M2, C2, B2)),
    Move = move(DM, DC, B2).

move(state(M, C, B), state(M2, C2, B2), Move) :-
    B = 1, B2 is 0, % Move from other side to original side
    member([DM, DC], [[0, 1], [0, 2], [1, 0], [1, 1], [2, 0]]), % Possible c
    M2 is M + DM, C2 is C + DC, valid(state(M2, C2, B2)),
    Move = move(DM, DC, B2).

% Solve the problem
solve(Node, Path, Moves, Goal) :-
    depthfirst(Node, Path, Moves, [Node], [], Goal).

depthfirst(Node, [Node], [], _, _, Goal) :-
    goal(Node, Goal).

%...more code
```