Performance Comparisons

Mingzhe Liu | 22306186

Instruction

This project is built on *PythonMazeSearch*[1] as maze generator and *MDP-with-Value-Iteration-and-Policy-Iteration*[2] as MDP algorithm prototype.
The main code is divided into three files: *GameSearch.py, solutions.py, mdp.py*.

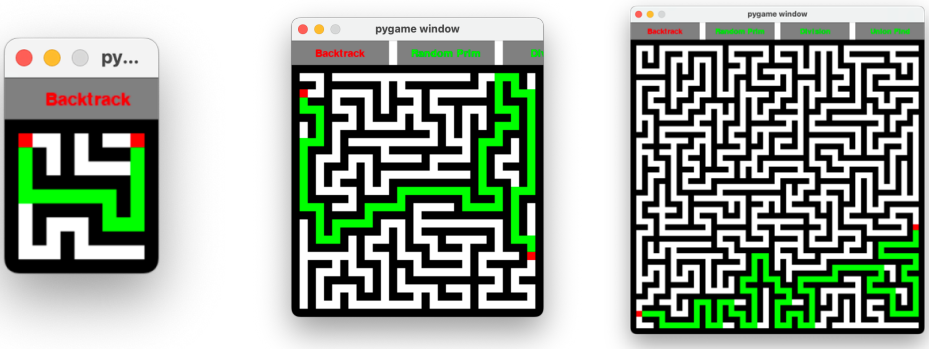
Maze size can be modified by changing *REC_WIDTH* and *REC_HEIGHT* in *GameSearch.py*.
GameSearch.py is the entry of the program. Start it by running the following command:

python GameSearch.py

After the program is started, mouse click button to choose one of the four maze generation algorithms.
Click any keyboard to do actions sequence: generate a maze, generate start and destination point, find a path, and clear the maze. Press again to run another algorithm.

A*, BFS, and DFS are implemented in *solutions.py*
Value iteration and policy iteration are implemented in *mdp.py*, parameters can be adjusted here.

The comparison between different search algorithms



The screenshots above are the mazes of three different sizes and the path solved by five algorithms (only solution). The following table is the comparison of all five algorithms using the loop count as metrics, which can stands for the time complexity of algorithms.

Since I want to compare the time complexity of different algorithms here, according to the control variable method, I didn't tune the value iteration and policy iteration for different parameters, the parameters they use are both:

REWARD = -0.01
DISCOUNT = 0.99
MAX_ERROR = 10^{-3}

Loop count comparison

Loop count	A*	BFS	DFS	VI	PI
11 * 11	27	42	31	31944	195086
31 * 31	123	171	285	814928	8441911
51 * 51	347	432	234	8146332	41382576
101 * 101	827	845	3760	-	-

To make the comparison more obvious, I added a 101*101 maze to compare the time complexity. Since the comparison of the last two algorithms is obvious enough, and the execution time of these two algorithms is too long for a maze of this size, I only compared the first three algorithms.

Conclusion:
From the above table, we can see that A* and BFS have similar and shortest execution time, DFS has slightly longer execution time, Value iteration has longer execution time, Policy iteration has the longest execution time, and the latter two have very fast-growing execution time as the maze gets bigger.

Space complexity comparison

Space complexity	A*	BFS	DFS	VI	PI
	$O(b^d)$	$O(b^d)$	$O(bd)$	$O(\|S\|)$	$O(\|S\|^2)$

b: Branching factor of the maze
d: The depth of the maze.
S: The number of states in the maze

The above table shows the space complexity of the five algorithms.

The space complexity of A* algorithm depends on the size of the open list and the closed list. In the worst-case scenario, where every node is added to both lists, the space complexity of A* algorithm is $O(b^d)$.

BFS algorithm explores all nodes at the current depth before moving on to nodes at the next depth level. In the worst-case scenario, where the solution is at the maximum depth, BFS will store all nodes at that depth in memory.

DFS algorithm explores as far as possible along each branch before backtracking. In the worst-case scenario, DFS could store all nodes along a single branch.

Value iteration requires storing the values of all the states in the search space, resulting in linear space complexity.

Policy iteration requires storing the values and policies for all the states in the search space.

In summary, the space complexity of A* algorithm, BFS, and DFS depend on the size of the maze, while the space complexity of value iteration and policy iteration depends on the number of states in the maze.

The comparison between different MDP algorithms

Maze size: 11 * 11

These are the parameters I chose for both MDP algorithms:

REWARD = -0.01
DISCOUNT = 0.8
MAX_ERROR = 10^{-3}

The current maze is small, and the effect of the parameters is not significant, so punishment and max_error I chose a relatively common value.
There are not so many steps to take, so I chose a relatively small value for discount to speed up the converge.

	Value iteration	**Policy iteration**
Iteration count	30	7
Convergence Rate	0.0175s	0.0288s

Maze size: 31 * 31

These are the parameters I chose for both MDP algorithms:

REWARD = -0.01
DISCOUNT = 0.99
MAX_ERROR = 10^{-3}

I set DISCOUNT to a larger value because when it was small, the score was prematurely reduced to very little, making it impossible to converge.

I first set the DISCOUNT to 0.9, and then the value iteration to get the results hit the wall, there was a bug in drawing the route, but at this time the policy iteration can be converged, so I got the following conclusion.

Stability of the solution measures how much the solution changes with a change in the discount factor. Value iteration can be unstable for high discount factors, while policy iteration is more stable

	Value iteration	**Policy iteration**
Iteration count	347	66
Convergence Rate	1.229s	9.614s
Stability	Unstable	Stable

Maze size: 51 * 51

The parameters haven't been changed from above.

	Value iteration	Policy iteration
Iteration count	577	116
Convergence Rate	5.252s	48.598s
Stability	Unstable	Stable

Conclusion:
As the size of the maze increases, the number of iterations of policy iteration is less, but policy iteration becomes slower than value iteration. And value iteration is more unstable.

The comparison between search and MDP algorithms

Search algorithms are used to find a path from a starting point to a destination in a maze, whereas MDP algorithms are used to solve problems in which the outcomes are uncertain and involve a trade-off between immediate and future rewards.

A* and BFS algorithms are guaranteed to find the shortest path in a maze, whereas DFS algorithm may not find the optimal path. Value iteration and policy iteration algorithms are guaranteed to converge to the optimal policy in an MDP.

Search algorithms assume that the maze is a graph with nodes and edges, whereas MDP algorithms assume that the problem can be modelled as a Markov Decision Process.

Search algorithms and MDP algorithms are used for different types of problems, and their efficiency and optimality depend on the specific problem being solved.

Justification for design choices

The heuristic I chose in A* is to calculate the Euclidean distance between two points. Because it is admissible and consistent. Admissible means that the heuristic never overestimates the actual distance to the goal, while consistent means that the estimated distance between any two adjacent nodes is less than or equal to the actual distance between those nodes. The Euclidean distance heuristic satisfies both of these properties, making it a good choice for A* algorithm.

Time and space complexity and stability were chosen as metrics because they are the most intuitive effects felt when the size of the maze gets larger.

Reference

[1] marblexu (2019) PythonMazeSearch [Source Code].
https://github.com/marblexu/PythonMazeSearch

[2] SparkShen02 (2021) MDP-with-Value-Iteration-and-Policy-Iteration [Source Code].
https://github.com/SparkShen02/MDP-with-Value-Iteration-and-Policy-Iteration

Appendix A*

```
class SearchEntry:
    def __init__(self, x, y, g_cost, f_cost=0, pre_entry=None):
        self.x = x
        self.y = y
        # cost move form start entry to this entry
        self.g_cost = g_cost
        self.f_cost = f_cost
        self.pre_entry = pre_entry

    def get_pos(self):
        return self.x, self.y


def a_star_search(maze, source, dest):
    def get_fast_position(open_list):
        fast = None
        for entry in open_list.values():
            if fast is None:
                fast = entry
            elif fast.f_cost > entry.f_cost:
                fast = entry
        return fast

    def get_positions(maze, location):
        offsets = [(-1, 0), (0, -1), (1, 0), (0, 1)]
        pos_list = []
        for offset in offsets:
            x, y = (location.x + offset[0], location.y + offset[1])
            if maze.isValid(x, y) and maze.isMovable(x, y):
                pos_list.append((x, y))
        return pos_list

    # check if the position is in list
    def is_in_list(list, pos):
        if pos in list:
            return list[pos]
        return None

    def cal_heuristic(pos, dest):
        return abs(dest.x - pos[0]) + abs(dest.y - pos[1])

    def get_move_cost(location, pos):
        if location.x != pos[0] and location.y != pos[1]:
            return 1.4
        else:
            return 1

    def add_adjacent_positions(maze, cur_loc, dest, open_list,
closed_list):
        pos_list = get_positions(maze, cur_loc)
        for pos in pos_list:
            # if position is already in closed_list, do nothing
            if is_in_list(closed_list, pos) is None:
                find_entry = is_in_list(open_list, pos)
                h_cost = cal_heuristic(pos, dest)
                g_cost = cur_loc.g_cost + get_move_cost(cur_loc, pos)
                if find_entry is None:
                    # if position is not in open_list, add it to open_list
```

```
                    open_list[pos] = SearchEntry(pos[0], pos[1], g_cost,
g_cost + h_cost, cur_loc)
                elif find_entry.g_cost > g_cost:
                    find_entry.g_cost = g_cost
                    find_entry.f_cost = g_cost + h_cost
                    find_entry.pre_entry = cur_loc

    open_list = {}
    closed_list = {}
    location = SearchEntry(source[0], source[1], 0.0)
    dest = SearchEntry(dest[0], dest[1], 0.0)
    open_list[source] = location
    counter = 0
    while True:
        counter += 1
        location = get_fast_position(open_list)
        if location is None:
            print("can't find valid path")
            break
        if location.x == dest.x and location.y == dest.y:
            break
        closed_list[location.get_pos()] = location
        open_list.pop(location.get_pos())
        add_adjacent_positions(maze, location, dest, open_list,
closed_list)
    while location is not None:
        maze.setMap(location.x, location.y, MAP_ENTRY_TYPE.MAP_PATH)
        location = location.pre_entry
    print('loop times: ', counter)
```

Appendix BFS

```
def bfs_search(maze, source, dest):
    # Define directions (up, down, left, right)
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    # Create a queue and add the source location
    queue = deque()
    queue.append(source)

    # Create a dictionary to store predecessors and mark source as visited
    predecessors = {}
    predecessors[source] = None
    visited = set()
    visited.add(source)

    counter = 0
    # Keep searching until the queue is empty or the destination is found
    while queue:
        counter += 1
        # Dequeue next position from the queue
        curr_pos = queue.popleft()

        # If destination is found, break
        if curr_pos == dest:
            break

        # Explore all neighboring positions
        for direction in directions:
            x, y = curr_pos[0] + direction[0], curr_pos[1] + direction[1]
```

```
            new_pos = x, y

            # Check if position is valid and not visited
            if maze.isValid(x, y) and maze.isMovable(x, y) and new_pos not
in visited:
                predecessors[new_pos] = curr_pos
                visited.add(new_pos)
                queue.append(new_pos)

    # If destination was not found, return None
    if dest not in predecessors:
        print("Can't find valid path.")
        return None

    # Mark the path from destination to source on the maze
    curr_pos = dest
    print('loop times: ', counter)
    while curr_pos is not None:
        maze.setMap(curr_pos[0], curr_pos[1], MAP_ENTRY_TYPE.MAP_PATH)
        curr_pos = predecessors[curr_pos]

    return maze
```

Appendix DFS

```
def dfs_search(maze, source, dest):
    # Define directions (up, down, left, right)
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    # Create a stack and add the source location
    stack = [source]

    # Create a dictionary to store predecessors and mark source as visited
    predecessors = {}
    predecessors[source] = None
    visited = set()
    visited.add(source)

    counter = 0
    # Keep searching until the stack is empty or the destination is found
    while stack:
        counter += 1
        # Pop the top position from the stack
        curr_pos = stack.pop()

        # If destination is found, break
        if curr_pos == dest:
            break

        # Explore all neighboring positions in reverse order
        for direction in directions[::-1]:
            x, y = curr_pos[0] + direction[0], curr_pos[1] + direction[1]
            new_pos = x, y

            # Check if position is valid and not visited
            if maze.isValid(x, y) and maze.isMovable(x, y) and new_pos not
in visited:
                predecessors[new_pos] = curr_pos
                visited.add(new_pos)
                stack.append(new_pos)
```

```
    # If destination was not found, return None
    if dest not in predecessors:
        print("Can't find valid path.")
        return None

    print('loop times: ', counter)

    # Mark the path from destination to source on the maze
    curr_pos = dest
    while curr_pos is not None:
        maze.setMap(curr_pos[0], curr_pos[1], MAP_ENTRY_TYPE.MAP_PATH)
        curr_pos = predecessors[curr_pos]

    return maze
```

Appendix MDP

```
import copy
import random

from memory_profiler import profile

from GameMap import Map, MAP_ENTRY_TYPE

REWARD_VALUE = -0.01  # constant reward for non-terminal states
DISCOUNT_VALUE = 0.99
MAX_ERROR_VALUE = 10 ** (-3)

REWARD_POLICY = -0.01  # constant reward for non-terminal states
DISCOUNT_POLICY = 0.99
MAX_ERROR_POLICY = 10 ** (-3)

# dx, dy
# Down, Left, Up, Right
ACTIONS = [(0, 1), (-1, 0), (0, -1), (1, 0)]


# Get the utility of the state reached by performing the given action from
the given state
def get_utility(maze, x, y, action):
    dx, dy = ACTIONS[action]
    new_x, new_y = x + dx, y + dy
    if not maze.isValid(new_x, new_y) or not maze.isMovable(new_x, new_y):
        return maze.map[y][x]
    else:
        return maze.map[new_y][new_x]


def calculate_utility(maze, x, y, action, reward, discount):
    u = reward
    u += 0.1 * discount * get_utility(maze, x, y, (action - 1) % 4)
    u += 0.8 * discount * get_utility(maze, x, y, action)
    u += 0.1 * discount * get_utility(maze, x, y, (action + 1) % 4)
    return u


def value_iteration(maze_origin: Map, dest):
    maze = copy.deepcopy(maze_origin)
    print('During the value iteration: \n')
```

```
    maze.showNumericMap(dest)
    counter = 0
    loop_counter = 0
    while True:
        next_maze = copy.deepcopy(maze_origin)
        error = 0
        for y in range(maze.height):
            for x in range(maze.width):
                loop_counter += 4
                # if () this is the wall/goal
                if not maze.isValid(x, y) or not maze.isMovable(x, y) or
(x, y) == dest:
                    continue
                # Bellman update
                next_maze.map[y][x] = max(
                    [calculate_utility(maze, x, y, action, REWARD_VALUE,
DISCOUNT_VALUE) for action in
                     range(len(ACTIONS))])
                # next_maze.setMap(x, y, max([calculate_utility(maze, x, y,
action) for action in range(len(ACTIONS))]))
                error = max(error, abs(next_maze.map[y][x] -
maze.map[y][x]))
        maze = next_maze
        maze.showNumericMap(dest)
        counter += 1
        print('Iteration count: ', counter)
        if error < MAX_ERROR_VALUE * (1 - DISCOUNT_VALUE) / DISCOUNT_VALUE:
            break
    print('Loop count: ', loop_counter)
    return maze


def get_optimal_policy(maze: Map, dest):
    policy = [[-1] * maze.width for _ in range(maze.height)]
    for y in range(maze.height):
        for x in range(maze.width):
            if not maze.isValid(x, y) or not maze.isMovable(x, y) or (x, y)
== dest:
                continue
            max_action, max_utility = None, -float('inf')
            for action in range(len(ACTIONS)):
                utility = calculate_utility(maze, x, y, action,
REWARD_VALUE, DISCOUNT_VALUE)
                if utility > max_utility:
                    max_action, max_utility = action, utility
            policy[y][x] = max_action
    return policy


def print_policy(maze: Map, policy: list, dest):
    res = ''
    for y in range(len(policy)):
        res += '|'
        for x in range(len(policy[0])):
            if not maze.isValid(x, y) or not maze.isMovable(x, y):
                val = '#'
            elif (x, y) == dest:
                val = 'GOAL'
            else:
                val = ["v", "<", "^", ">"][policy[y][x]]
            res += " " + val[:5].ljust(5) + " |"  # format
```

```
        res += '\n'
    print(res)


def draw_policy_on_maze(maze: Map, policy: list, source, dest):
    x = source[0]
    y = source[1]
    while (x, y) != dest:
        x, y = x + ACTIONS[policy[y][x]][0], y + ACTIONS[policy[y][x]][1]
        maze.setMap(x, y, MAP_ENTRY_TYPE.MAP_PATH)
    print('Draw path finished')


def policy_evaluation(policy, maze_origin: Map, dest, loop_counter):
    maze = copy.deepcopy(maze_origin)
    while True:
        next_maze = copy.deepcopy(maze_origin)
        error = 0
        for y in range(maze.height):
            for x in range(maze.width):
                loop_counter += 1
                if not maze_origin.isValid(x, y) or not
maze_origin.isMovable(x, y) or (x, y) == dest:
                    continue
                next_maze.map[y][x] = calculate_utility(maze, x, y,
policy[y][x], REWARD_POLICY, DISCOUNT_POLICY)
                error = max(error, abs(next_maze.map[y][x] -
maze.map[y][x]))
        maze = next_maze
        if error < MAX_ERROR_POLICY * (1 - DISCOUNT_VALUE) /
DISCOUNT_VALUE:
            break
    return maze, loop_counter


def policy_iteration(maze: Map, dest):
    policy = [[random.randint(0, 3) for _ in range(maze.width)] for _ in
              range(maze.height)]  # construct a random policy
    print("During the policy iteration:\n")
    counter = 0
    loop_counter = 0
    while True:
        counter += 1
        maze, loop_counter = policy_evaluation(policy, maze, dest,
loop_counter)
        unchanged = True
        for y in range(maze.height):
            for x in range(maze.width):
                if not maze.isValid(x, y) or not maze.isMovable(x, y) or
(x, y) == dest:
                    continue
                max_action, max_utility = None, -float('inf')
                for action in range(len(ACTIONS)):
                    loop_counter += 1
                    utility = calculate_utility(maze, x, y, action,
REWARD_POLICY, DISCOUNT_POLICY)
                    if utility > max_utility:
                        max_action, max_utility = action, utility
                if max_utility > calculate_utility(maze, x, y,
policy[y][x], REWARD_POLICY, DISCOUNT_POLICY):
                    policy[y][x] = max_action
```

```
                unchanged = False
    if unchanged:
        break
    print_policy(maze, policy, dest)
    print('Iteration times: ', counter)
print('Loop count: ', loop_counter)
return policy
```