Search Algorithims Analysis

Uninformed Search Algorithims

```
# Node code
from sys import setrecursionlimit
setrecursionlimit(100000)

class Node:
    def __init__(self, state, parent=None):
        self.state = [row[:] for row in state]
        self.parent = parent  # tracing

    def __str__(self):
        return '\n'.join([' '.join(row) for row in self.state])

class PuzzleSolver:
    def __init__(self, start, goal=None):
        self.start = start
        self.goal = goal if goal else Node([['1', '2', '3'], ['4', '5', '6'], ['7', '8', ' ']])

    def is_solvable(self, state):
        flat = [tile for row in state.state for tile in row if tile != ' ']
        inversions = 0
        for i in range(len(flat)):
            for j in range(i + 1, len(flat)):
                if flat[i] > flat[j]:
                    inversions += 1
        return inversions % 2 == 0

    def find_space(self, state):
        for i in range(3):
            for j in range(3):
                if state.state[i][j] == ' ':
                    return (i, j)
        return None

    def find_moves(self, pos):
        x, y = pos
        return [(x + 1, y), (x - 1, y), (x, y + 1), (x, y - 1)]

    def is_valid(self, move):
        x, y = move
        return 0 <= x < 3 and 0 <= y < 3

    def play_move(self, state, move, space):
        x, y = space
        new_x, new_y = move
        new_state = [row[:] for row in state.state]
        new_state[x][y], new_state[new_x][new_y] = new_state[new_x][new_y], new_state[x][y]
        return Node(new_state, state)

    def generate_children(self, state):
        children = []
        space = self.find_space(state)
        moves = self.find_moves(space)
        for move in moves:
            if self.is_valid(move):
                child = self.play_move(state, move, space)
                children.append(child)
        return children

    def state_to_string(self, state):
        return ''.join([''.join(row) for row in state.state])
```

3. Backtracking

Introduction: Backtracking is a refined form of DFS that incrementally builds candidates to the solution and abandons a path as soon as it determines it cannot lead to a valid solution.

- **Pros:**
  - Efficient memory usage (only stores the current path).
  - Can prune invalid paths early, reducing unnecessary exploration.
- **Cons:**
  - Not optimal: may not find the shortest solution.
  - Requires careful design of pruning conditions specific to the 8-puzzle.
  - Can still be slow if many paths need exploration.

```
# BAcktracking solver
class BacktrackingSolver(PuzzleSolver):
    def solve(self):
        def backtrack(node):
            state_list = [self.state_to_string(self.start)]
            def recursive_backtrack(current):
                if self.state_to_string(current) == self.state_to_string(self.goal):
                    return current
                for child in self.generate_children(current):
                    child_str = self.state_to_string(child)
                    if child_str not in state_list:
                        state_list.append(child_str)
                        result = recursive_backtrack(child)
                        if result:
                            return result
                return None
            return recursive_backtrack(node)
        final_state = backtrack(self.start)
        if final_state:
            print("Backtracking Solution Found:")
            self.disp_solution(final_state)
        else:
            print("No solution found with backtracking.")
    def disp_solution(self, final_state):
        # Display the solution path from start to goal
        if not final_state:
            print("No path to display.")
            return
        path = []
        current = final_state
        while current:
            path.append(current)
            current = current.parent
        path.reverse()
        for i, node in enumerate(path):
            print(f"Step {i}:\n{node}\n")
```

2. Depth-First Search (DFS)

Introduction: DFS explores as far as possible along each branch before backtracking, using a stack (explicit or via recursion) to manage the search.

- **Pros:**
  - Low memory usage (linear space complexity: O(bm), where m is the maximum depth).
  - Can find a solution quickly if it lies on an early branch.
- **Cons:**
  - Not optimal: may find a longer path than necessary.
  - Not complete in infinite state spaces (can get stuck in loops without cycle detection).
  - Unpredictable performance for the 8-puzzle due to variable depths.

```
# DFS solver
class DFSSolver(PuzzleSolver):
    def solve(self):
        open_list = [self.start]
        closed_list = set()

        while open_list:
            current = open_list.pop()
            current_str = self.state_to_string(current)
```

```
            if current_str == self.state_to_string(self.goal):
                print("DFS Solution Found:")
                self.disp_solution(current)
                return

            if current_str not in closed_list:
                closed_list.add(current_str)
                children = self.generate_children(current)
                open_list.extend(children)
        print("No solution found with DFS.")
    def disp_solution(self, final_state):
        # Display the solution path from start to goal
        if not final_state:
            print("No path to display.")
            return
        path = []
        current = final_state
        while current:
            path.append(current)
            current = current.parent
        path.reverse()
        for i, node in enumerate(path):
            print(f"Step {i}:\n{node}\n")
```

1. Breadth-First Search (BFS)

Introduction: BFS explores all nodes at the present depth level before moving to nodes at the next depth level, using a queue to maintain the frontier of nodes to be explored.

- **Pros:**
  - Guarantees the shortest path to the solution (optimal) in terms of moves.
  - Complete: will find a solution if one exists.
- **Cons:**
  - High memory usage due to storing all nodes at each level (exponential space complexity: O(b^d)).
  - Can be slow for deep solutions in large state spaces.

```
# BFS solver
class BFSSolver(PuzzleSolver):
    def solve(self):
        open_list = [self.start]
        closed_list = set()

        while open_list:
            current = open_list.pop(0)
            current_str = self.state_to_string(current)

            if current_str == self.state_to_string(self.goal):
                print("BFS Solution Found:")
                self.disp_solution(current)
                return

            if current_str not in closed_list:
                closed_list.add(current_str)
                children = self.generate_children(current)
                open_list.extend(children)
        print("No solution found with BFS.")
    def disp_solution(self, final_state):
        # Display the solution path from start to goal
        if not final_state:
            print("No path to display.")
            return
        path = []
        current = final_state
        while current:
            path.append(current)
            current = current.parent
        path.reverse()
        for i, node in enumerate(path):
            print(f"Step {i}:\n{node}\n")
```

4. Depth-First Iterative Deepening (DFID)

Introduction: DFID combines BFS's optimality with DFS's space efficiency by running DFS with increasing depth limits until a solution is found.

- **Pros:**
  - Optimal: finds the shortest solution like BFS.
  - Memory-efficient (O(bd) space, where d is the solution depth).
  - Complete if a solution exists.
- **Cons:**
  - Redundant computation: re-explores shallower nodes multiple times.
  - Slower than BFS in terms of time due to repeated exploration.

```
# DFID solver
class DFIDSolver(PuzzleSolver):
    def solve(self):
        def dls(node, depth, visited):
            if depth < 0:
                return None
            current_str = self.state_to_string(node)
            if current_str == self.state_to_string(self.goal):
                return node
            if current_str in visited:
                return None
            visited.add(current_str)
            for child in self.generate_children(node):
                result = dls(child, depth - 1, visited.copy())
                if result:
                    return result
            return None
        depth = 0
        while True:
            visited = set()
            result = dls(self.start, depth, visited)
            if result:
                print("DFID Solution Found:")
                self.disp_solution(result)
                return
            depth += 1
            if depth > 50:
                print("No solution found with DFID within depth limit.")
                return
    def disp_solution(self, final_state):
        # Display the solution path from start to goal
        if not final_state:
            print("No path to display.")
            return
        path = []
        current = final_state
        while current:
            path.append(current)
            current = current.parent
        path.reverse()
        for i, node in enumerate(path):
            print(f"Step {i}:\n{node}\n")



def main():

    start = Node([['1', '2', '3'], [' ', '5', '6'], ['4', '7', '8']])

    print("Starting State:")
    print(start)

    print("\nGoal State:")
    goal = Node([['1', '2', '3'], ['4', '5', '6'], ['7', '8', ' ']])
    print(goal)

    print("\nSolving with Backtracking:")
    backtracking_solver = BacktrackingSolver(start=start, goal=goal)
    backtracking_solver.solve()
```

```
    print("\nSolving with DFS:")
    dfs_solver = DFSSolver(start=start, goal=goal)
    dfs_solver.solve()

    print("\nSolving with BFS:")
    bfs_solver = BFSSolver(start=start, goal=goal)
    bfs_solver.solve()

    print("\nSolving with DFID:")
    dfid_solver = DFIDSolver(start=start, goal=goal)
    dfid_solver.solve()

main()
```

```
import timeit
from memory_profiler import memory_usage

# Main function
def profile_solver(solver_class, start, goal):
    solver = solver_class(start=start, goal=goal)
    start_time = timeit.default_timer()
    mem_usage = memory_usage((solver.solve,))
    end_time = timeit.default_timer()
    return end_time - start_time, max(mem_usage) - min(mem_usage)

start = Node([['1', '2', '3'], [' ', '5', '6'], ['4', '7', '8']])
goal = Node([['1', '2', '3'], ['4', '5', '6'], ['7', '8', ' ']])

print("Starting State:")
print(start)

print("\nGoal State:")
print(goal)

print("\nProfiling Backtracking Solver:")
time_bt, mem_bt = profile_solver(BacktrackingSolver, start, goal)
print(f"Time: {time_bt} seconds, Memory: {mem_bt} MiB")
print("\nProfiling DFS Solver:")
time_dfs, mem_dfs = profile_solver(DFSSolver, start, goal)
print(f"Time: {time_dfs} seconds, Memory: {mem_dfs} MiB")
print("\nProfiling BFS Solver:")
time_bfs, mem_bfs = profile_solver(BFSSolver, start, goal)
print(f"Time: {time_bfs} seconds, Memory: {mem_bfs} MiB")

print("\nProfiling DFID Solver:")
time_dfid, mem_dfid = profile_solver(DFIDSolver, start, goal)
print(f"Time: {time_dfid} seconds, Memory: {mem_dfid} MiB")
```

Informed Search Algorithims

```
import heapq
import timeit
from memory_profiler import memory_usage

# Priority Queue for managing nodes
class PriorityQueue:
    def __init__(self):
        self.heap = []

    def enqueue(self, x):
        heapq.heappush(self.heap, x)

    def dequeue(self):
        return heapq.heappop(self.heap)

    def is_empty(self):
        return len(self.heap) == 0

# Node class for representing puzzle states
class Node:
```

```python
    def __init__(self, state, parent=None):
        self.state = tuple(tuple(row) for row in state)  # Immutable state
        self.parent = parent
        self.g = 0  # Cost from start
        self.h = 0  # Heuristic estimate
        self.f = 0  # Total estimated cost (g + h)

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

    def heuristic(self, goal):
        """Manhattan distance heuristic"""
        distance = 0
        goal = tuple(tuple(row) for row in goal)
        for r in range(3):
            for c in range(3):
                val = self.state[r][c]
                if val != 0:
                    goal_row, goal_col = divmod(val - 1, 3)
                    distance += abs(r - goal_row) + abs(c - goal_col)
        return distance

    def out_place(self, goal):
        """Out-of-place heuristic"""
        distance = 0
        goal = tuple(tuple(row) for row in goal)
        for r in range(3):
            for c in range(3):
                val = self.state[r][c]
                if val != 0:
                    goal_row, goal_col = divmod(val - 1, 3)
                    if r != goal_row or c != goal_col:
                        distance += 1
        return distance

    def to_list(self):
        """Convert tuple state to list of lists"""
        return [list(row) for row in self.state]

# Base Puzzle Solver class
class PuzzleSolver:
    def __init__(self, start, goal):
        self.start = start  # List of lists
        self.goal = goal    # List of lists

    def is_solvable(self):
        """Check if the puzzle is solvable based on inversions"""
        flat = [x for row in self.start for x in row if x != 0]
        inversions = 0
        for i in range(len(flat)):
            for j in range(i + 1, len(flat)):
                if flat[i] > flat[j]:
                    inversions += 1
        return inversions % 2 == 0

    def find_space(self, state):
        """Find the position of the blank tile (0)"""
        for r in range(3):
            for c in range(3):
                if state[r][c] == 0:
                    return (r, c)
        return None

    def find_moves(self, pos):
        """Possible moves from the blank tile position"""
        (r, c) = pos
        return [(r-1, c), (r+1, c), (r, c-1), (r, c+1)]

    def is_valid(self, move):
        """Check if a move is within the 3x3 grid"""
        (r, c) = move
        return 0 <= r < 3 and 0 <= c < 3

    def play_move(self, state, move, space):
        """Execute a move by swapping the blank tile"""
        (r, c) = space
        (new_r, new_c) = move
```

```
        new_state = [row[:] for row in state]
        new_state[r][c], new_state[new_r][new_c] = new_state[new_r][new_c], new_state[r][c]
        return new_state

    def find_children(self, state):
        """Generate all possible child states"""
        children = []
        space = self.find_space(state)
        for move in self.find_moves(space):
            if self.is_valid(move):
                child_state = self.play_move(state, move, space)
                children.append(child_state)
        return children
```

6. Best-First Search

Introduction: Best-First Search is a greedy, informed search that prioritizes nodes based solely on a heuristic function (h(n)), without considering the cost from the start (g(n)).

- **Pros:**
  - Can be fast if the heuristic closely aligns with the solution path.
  - Simple to implement with a priority queue.
- **Cons:**
  - Not optimal: may find a suboptimal solution due to greediness.
  - Not complete: can get trapped in local optima or infinite branches.
  - Memory usage can still be high (O(b^d)).

```
# Best-First Search Solver
class PuzzleSolverWithBestFS(PuzzleSolver):
    def solve_puzzle(self):
        if not self.is_solvable():
            return None, "Puzzle is not solvable"
        pq = PriorityQueue()
        start_node = Node(self.start)
        start_node.h = start_node.out_place(self.goal)
        start_node.f = start_node.h
        pq.enqueue(start_node)
        explored = set()
        while not pq.is_empty():
            current_node = pq.dequeue()
            if current_node.state == tuple(tuple(row) for row in self.goal):
                return self.reconstruct_path(current_node), f"Solution found in {current_node.g} moves"
            state_str = current_node.state
            if state_str in explored:
                continue
            explored.add(state_str)
            for child_state in self.find_children(current_node.to_list()):
                child_tuple = tuple(tuple(row) for row in child_state)
                if child_tuple not in explored:
                    child_node = Node(child_state, current_node)
                    child_node.g = current_node.g + 1
                    child_node.h = child_node.out_place(self.goal)
                    child_node.f = child_node.h
                    pq.enqueue(child_node)
        return None, "No solution found"

    def reconstruct_path(self, node):
        """Reconstruct the solution path"""
        path = []
        while node is not None:
            path.append(node.to_list())
            node = node.parent
        path.reverse()
        return path
```

5. A* Search

Introduction: A* is an informed search algorithm that uses a heuristic function (e.g., Manhattan distance) to guide the search toward the goal, combining the cost to reach a node (g(n)) and the estimated cost to the goal (h(n)).

- **Pros:**
  - Optimal: finds the shortest solution if the heuristic is admissible (e.g., Manhattan distance).
  - Efficient: heuristic reduces the number of nodes explored.
  - Complete with a good heuristic.
- **Cons:**
  - Memory-intensive (O(b^d) space complexity).
  - Performance depends heavily on the quality of the heuristic.

```
# A* Search Solver
class PuzzleSolverWithAStar(PuzzleSolver):
    def solve_puzzle(self):
        if not self.is_solvable():
            return None, "Puzzle is not solvable"
        pq = PriorityQueue()
        start_node = Node(self.start)
        start_node.h = start_node.heuristic(self.goal)
        start_node.f = start_node.g + start_node.h
        pq.enqueue(start_node)
        explored = set()
        while not pq.is_empty():
            current_node = pq.dequeue()
            if current_node.state == tuple(tuple(row) for row in self.goal):
                return self.reconstruct_path(current_node), f"Solution found in {current_node.g} moves"
            state_str = current_node.state
            if state_str in explored:
                continue
            explored.add(state_str)
            for child_state in self.find_children(current_node.to_list()):
                child_tuple = tuple(tuple(row) for row in child_state)
                if child_tuple not in explored:
                    child_node = Node(child_state, current_node)
                    child_node.g = current_node.g + 1
                    child_node.h = child_node.heuristic(self.goal)
                    child_node.f = child_node.g + child_node.h
                    pq.enqueue(child_node)
        return None, "No solution found"

    def reconstruct_path(self, node):
        """Reconstruct the solution path"""
        path = []
        while node is not None:
            path.append(node.to_list())
            node = node.parent
        path.reverse()
        return path


# Main function

def main():

    start1 = [['1', '2', '3'], ['4', '5', '6'], ['0', '7', '8']]
    ps1 = PuzzleSolverWithAStar(start1)
    solution1, message1 = ps1.solve_puzzle()
    print("Test 1:")
    print(message1)
    if solution1:
        for i, state in enumerate(solution1):
            print(f"Step {i}: {state}")
    print()

    start2 = [['1', '2', '3'], ['4', '5', '6'], ['0', '7', '8']]
    ps2 = PuzzleSolverWithBestFS(start2)
    solution2, message2 = ps2.solve_puzzle()
    print("Test 2:")
    print(message2)
    if solution2:
        for i, state in enumerate(solution2):
            print(f"Step {i}: {state}")
```

```
main()


# Profiling function
def profile_solver(solver_class, start, goal):
    solver = solver_class(start=start, goal=goal)
    start_time = timeit.default_timer()
    mem_usage = memory_usage((solver.solve_puzzle,))
    end_time = timeit.default_timer()
    return end_time - start_time, max(mem_usage) - min(mem_usage)

# Define start and goal states as lists of lists
start_list = [['1', '2', '3'], ['4', '5', '6'], ['0', '7', '8']]
goal_list = [['1', '2', '3'], ['4', '5', '6'], ['7', '8', '0']]

# Print states
print("Starting State:")
for row in start_list:
    print(row)

print("\nGoal State:")
for row in goal_list:
    print(row)

#Profile Best-First Search
print("\nPuzzleSolver With Best First Search:")
time_bfs, mem_bfs = profile_solver(PuzzleSolverWithBestFS, start_list, goal_list)
print(f"Time: {time_bfs} seconds, Memory: {mem_bfs} MiB")

#Profile A* Search
print("\nPuzzleSolver With AStar:")
time_astar, mem_astar = profile_solver(PuzzleSolverWithAStar, start_list, goal_list)
print(f"Time: {time_astar} seconds, Memory: {mem_astar} MiB")
```

Analysis of Uninformed and Informed Algorithims

	Uninformed				Informed	
	Backtracking	**DFS**	**DFID**	**BFS**	**A***	**Best FS**
Steps	31	431	3	3	3	3
Time	2.42137599997222	2.84015339985489	2.67866000020876	2.60968510014936	3.14628889993764	3.02687190007418
Memory	0.23828125	0.5546875	0.13671875	0.11328125	0.0078125	0.12109375

Note: The A* and Best Firist Seaarrch Algorithims worksk nealy same in differrent heuristics