**Batch: A-3 Roll No.: 16010122104**

**Experiment / assignment / tutorial No. 4**

**Grade: AA / AB / BB / BC / CC / CD /DD**

**Signature of the Staff In-charge with date**

|  |
| --- |
| **Title:** Implementation of uninformed search algorithms – BFS,DFS, DLS for the given problem |

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Objective:** Examine the efficiency and performance of uninformed search algorithms in solving various problems **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Expected Outcome of Experiment:**

|  |  |
| --- | --- |
| **Course Outcome** | **After successful completion of the course students should be able to** |
| **CO2** | Analyse and solve problems for goal based agent architecture (searching and planning algorithms) |

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Books/ Journals/ Websites referred:**

1. **“Artificial Intelligence: a Modern Approach” by Russell and Norving, Pearson education Publications**
2. **“Artificial Intelligence” By Rich and knight, Tata Mcgraw Hill Publications**
3. [**http://people.cs.pitt.edu/~milos/courses/cs2710/lectures/Class4.pdf**](http://people.cs.pitt.edu/~milos/courses/cs2710/lectures/Class4.pdf)
4. [**http://cs.williams.edu/~andrea/cs108/Lectures/InfSearch/infSearch.html**](http://cs.williams.edu/~andrea/cs108/Lectures/InfSearch/infSearch.html)
5. **http://www.cs.mcgill.ca/~dprecup/courses/AI/Lectures/ai-lecture02.pdf** [**http://homepage.cs.uiowa.edu/~hzhang/c145/notes/04a-search.pdf**](http://homepage.cs.uiowa.edu/~hzhang/c145/notes/04a-search.pdf)
6. [**http://wiki.answers.com/Q/Informed\_search\_techniques\_and\_uninformed\_search\_techniques**](http://wiki.answers.com/Q/Informed_search_techniques_and_uninformed_search_techniques)

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Pre Lab/ Prior Concepts:**

Problem solving, state-space trees, problem formulation, goal based agent architecture

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Historical Profile:**

### Problem-Solving Agent

A problem-solving agent is designed to find solutions to well-defined problems. This agent typically follows these steps:

1. **Formulate the Problem**: Define the initial state, goal state, and possible actions.
2. **Search for a Solution**: Use an appropriate search strategy to explore the problem space.
3. **Execute the Solution**: Apply the sequence of actions derived from the search.

### Uninformed Search Algorithms

Uninformed search algorithms, also known as blind search algorithms, are basic search strategies that explore the search space without any additional information about the goal's location beyond what is provided in the problem definition.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**New Concepts to be learned:**

Uninformed (blind) search.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Uninformed Searching Technique**

 B**readth-First Search (BFS)**:

* Explores all nodes at the present depth level before moving on to nodes at the next depth level.
* Complete and optimal if the cost is uniform.

 **Depth-First Search (DFS)**:

* Explores as far as possible along each branch before backtracking.
* Not complete or optimal, but requires less memory.

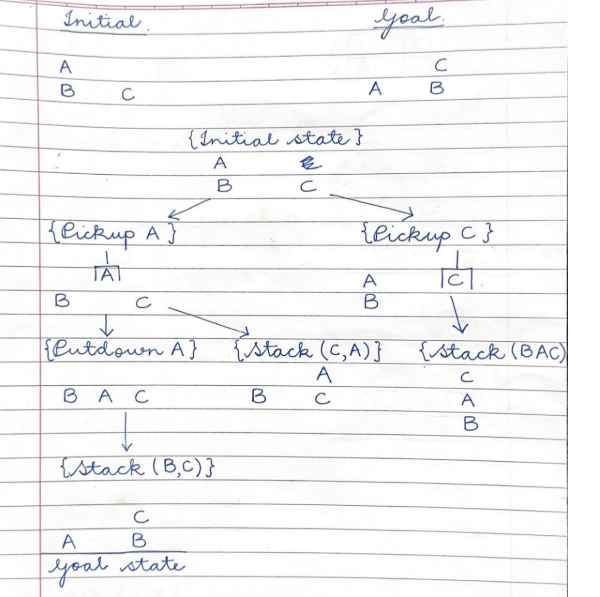
 **Depth-Limited Search (DLS)**:

* Depth-first search with a predetermined depth limit.
* Can overcome infinite path problems.

**Problem Statement:**

The Blocks World problem involves arranging uniquely labeled blocks from an initial configuration to a desired goal configuration, where blocks can be stacked on each other or placed directly on a table surface. A valid move consists of relocating only one block at a time, and only the topmost block of any stack can be moved to either the table or on top of another block. Your task is to implement a Depth Limited Search (DLS) algorithm that finds a sequence of valid moves transforming the initial state to the goal state while preventing infinite exploration by imposing a maximum search depth; the algorithm should either return the optimal sequence of moves or indicate that no solution exists within the specified depth limit.

**State-space tree :**

****

**Solution of chosen algorithm on the state-space tree:** *(Attach diagrams/solutions here)*

**depth\_limit = max depth to search to**

**agenda = [initial state]**

**if initial state is goal state then**

**return solution**

**else**

**while agenda not empty do**

**take node from front of agenda**

**if depth(node) < depth limit then**

**new nodes = apply operations to node**

**add new nodes to front of agenda**

**if goal state in new nodes then**

**return solution**

**Source code for the assigned problem with assigned algorithm:**

class BlocksWorld:

    def \_\_init\_\_(self, initial\_state, goal\_state):

        """

        Initialize a Blocks World problem.

        Each state is represented as a list of stacks (lists of blocks).

        """

        self.initial\_state = initial\_state

        self.goal\_state = goal\_state

    def is\_goal(self, state):

        """Check if the current state matches the goal state."""

        # Convert to a canonical representation for comparison

        canonical\_state = sorted([tuple(stack) for stack in state if stack])

        canonical\_goal = sorted([tuple(stack) for stack in self.goal\_state if stack])

        return canonical\_state == canonical\_goal

    def get\_valid\_moves(self, state):

        """

        Get all valid moves from the current state.

        A move is represented as (block, destination) where destination

        is either another block or "Table".

        """

        moves = []

        # Get all blocks that are at the top of stacks (can be moved)

        movable\_blocks = []

        for stack in state:

            if stack:  # If stack is not empty

                movable\_blocks.append(stack[-1])

        # For each movable block, find all valid destinations

        for block in movable\_blocks:

            # Can move to table

            moves.append((block, "Table"))

            # Can move to top of other stacks

            for target in movable\_blocks:

                if target != block:

                    moves.append((block, target))

        return moves

    def apply\_move(self, state, move):

        """

        Apply a move to the state and return the new state.

        Move is (block, destination).

        """

        block, destination = move

        new\_state = [stack[:] for stack in state]  # Deep copy

        # Find the stack containing the block

        source\_stack\_idx = None

        for i, stack in enumerate(new\_state):

            if stack and stack[-1] == block:

                source\_stack\_idx = i

                break

        if source\_stack\_idx is None:

            return None  # Block not found or not at top of any stack

        # Remove the block from its source stack

        new\_state[source\_stack\_idx].pop()

        # If the source stack is now empty, remove it

        if not new\_state[source\_stack\_idx]:

            new\_state.pop(source\_stack\_idx)

        # Place the block at its destination

        if destination == "Table":

            # Create a new stack with just the block

            new\_state.append([block])

        else:

            # Find the stack with the destination block on top

            dest\_stack\_idx = None

            for i, stack in enumerate(new\_state):

                if stack and stack[-1] == destination:

                    dest\_stack\_idx = i

                    break

            if dest\_stack\_idx is None:

                return None  # Destination block not found or not at top

            # Add the block to the destination stack

            new\_state[dest\_stack\_idx].append(block)

        return new\_state

    def depth\_limited\_search(self, limit):

        """

        Perform depth-limited search to find a solution.

        Returns a list of moves if a solution is found, None otherwise.

        """

        def state\_to\_tuple(s):

            """Convert a state to a hashable tuple for visited set."""

            return tuple(tuple(stack) for stack in s if stack)

        visited = set()

        def dls\_recursive(state, depth, path):

            # Check if we've reached the goal

            if self.is\_goal(state):

                return path

            # Check if we've reached the depth limit

            if depth >= limit:

                return None

            # Convert state to tuple for checking visited

            state\_tuple = state\_to\_tuple(state)

            if state\_tuple in visited:

                return None

            visited.add(state\_tuple)

            # Try all valid moves

            for move in self.get\_valid\_moves(state):

                new\_state = self.apply\_move(state, move)

                if new\_state:  # If the move is valid

                    result = dls\_recursive(new\_state, depth + 1, path + [move])

                    if result:

                        return result

            return None

        return dls\_recursive(self.initial\_state, 0, [])

def parse\_state\_input(state\_str):

    """Parse a string representation of a state."""

    state = []

    for stack\_str in state\_str.split(';'):

        if stack\_str:

            stack = stack\_str.split(',')

            state.append(stack)

    return state

def display\_solution(solution):

    """Display the solution steps."""

    if not solution:

        print("No solution found.")

        return

    print(f"Solution found with {len(solution)} moves:")

    for i, move in enumerate(solution, 1):

        block, destination = move

        print(f"Step {i}: Move block {block} to {'the table' if destination == 'Table' else 'block ' + destination}")

def display\_state(state):

    """Display a state in a readable format."""

    print("Current state:")

    for i, stack in enumerate(state, 1):

        print(f"Stack {i}: {stack}")

    print()

def interactive\_mode():

    """Run the program in interactive mode."""

    print("=== Blocks World Depth-Limited Search ===")

    print("Enter the initial state and goal state in the format: A,B;C,D")

    print("where each letter is a block and semicolons separate stacks.")

    print("Example: A,B;C represents two stacks [A,B] and [C]")

    # Get initial state

    initial\_str = input("\nEnter initial state: ")

    initial\_state = parse\_state\_input(initial\_str)

    # Get goal state

    goal\_str = input("Enter goal state: ")

    goal\_state = parse\_state\_input(goal\_str)

    # Get depth limit

    try:

        depth\_limit = int(input("Enter depth limit (default 10): ") or "10")

    except ValueError:

        depth\_limit = 10

        print("Invalid input. Using default depth limit of 10.")

    # Create the problem and solve

    problem = BlocksWorld(initial\_state, goal\_state)

    print("\nInitial state:")

    display\_state(initial\_state)

    print("Goal state:")

    display\_state(goal\_state)

    print(f"\nSearching for solution with depth limit {depth\_limit}...")

    solution = problem.depth\_limited\_search(depth\_limit)

    # Display the solution

    print("\nResults:")

    if solution:

        display\_solution(solution)

    else:

        print(f"No solution found within depth limit {depth\_limit}.")

        print("Try increasing the depth limit or check if the problem is solvable.")

def example\_mode():

    """Run a predefined example."""

    # Sussman anomaly

    initial\_state = [["A"], ["B", "C"]]

    goal\_state = [["B", "A", "C"]]

    print("=== Running Blocks World DLS with Sussman Anomaly Example ===")

    print("\nInitial state:")

    display\_state(initial\_state)

    print("Goal state:")

    display\_state(goal\_state)

    problem = BlocksWorld(initial\_state, goal\_state)

    depth\_limit = 10

    print(f"\nSearching for solution with depth limit {depth\_limit}...")

    solution = problem.depth\_limited\_search(depth\_limit)

    # Display the solution

    print("\nResults:")

    if solution:

        display\_solution(solution)

    else:

        print(f"No solution found within depth limit {depth\_limit}.")

def command\_line\_mode(args):

    """Run the program with command line arguments."""

    if len(args) != 4 or args[1] != 'dls':

        print("Usage: python blocks\_world\_dls.py dls <initial\_state> <goal\_state>")

        print("Example: python blocks\_world\_dls.py dls 'A,B;C' 'A;B;C'")

        return

    initial\_str = args[2]

    goal\_str = args[3]

    initial\_state = parse\_state\_input(initial\_str)

    goal\_state = parse\_state\_input(goal\_str)

    problem = BlocksWorld(initial\_state, goal\_state)

    depth\_limit = 10

    print("\nInitial state:")

    display\_state(initial\_state)

    print("Goal state:")

    display\_state(goal\_state)

    print(f"\nSearching for solution with depth limit {depth\_limit}...")

    solution = problem.depth\_limited\_search(depth\_limit)

    # Display the solution

    print("\nResults:")

    if solution:

        display\_solution(solution)

    else:

        print(f"No solution found within depth limit {depth\_limit}.")

def main():

    import sys

    if len(sys.argv) == 1:

        # No command line arguments, run interactive mode

        interactive\_mode()

    elif len(sys.argv) == 2 and sys.argv[1] == 'example':

        # Run example mode

        example\_mode()

    else:

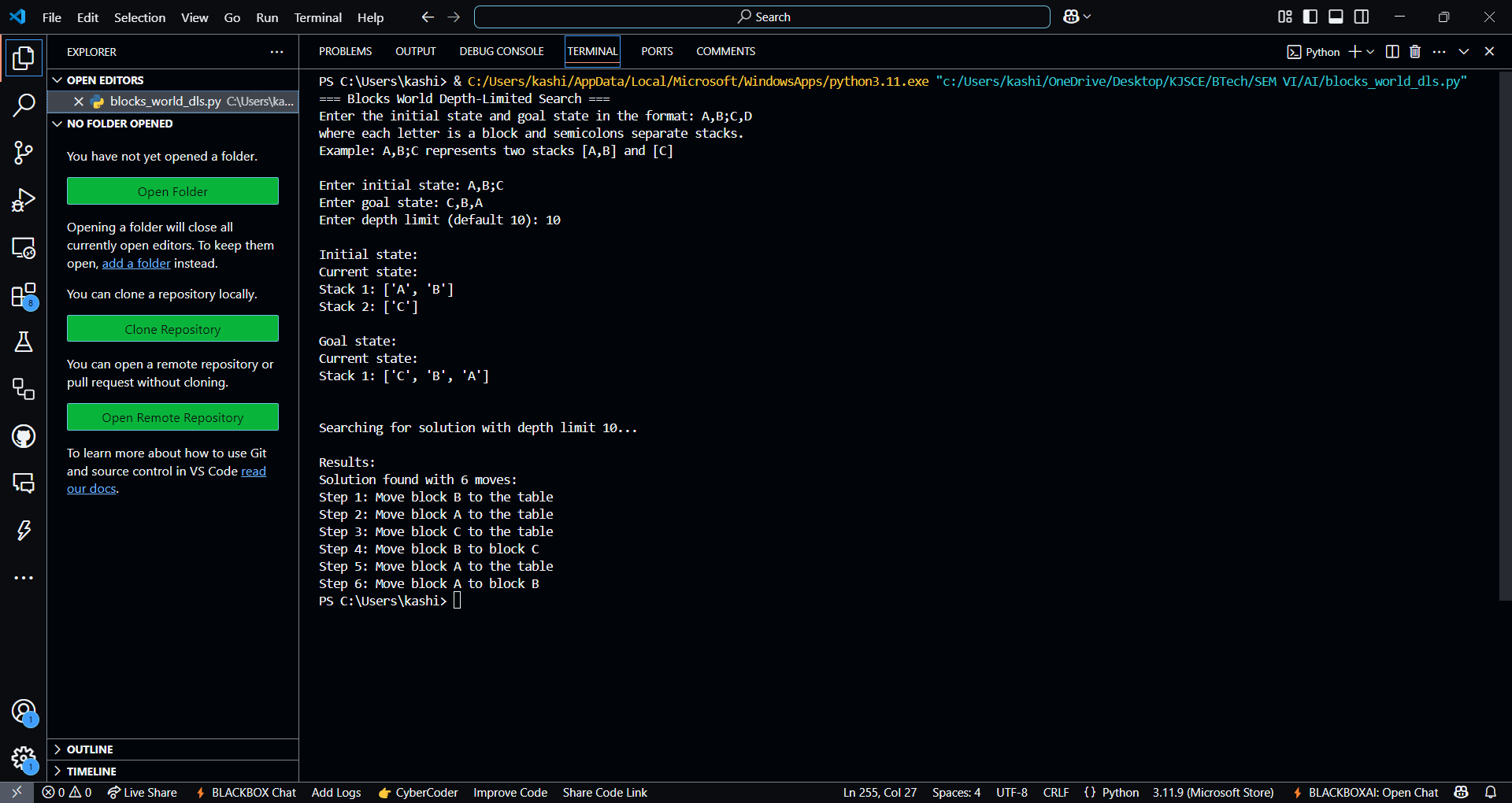
        # Run with command line arguments

        command\_line\_mode(sys.argv)

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**Output screenshots:**



**Comparison of performance of uninformed Algorithm:**

Key Algorithms and Characteristics

Breadth-First Search (BFS)

* Complete and optimal for unweighted graphs
* Time complexity: O(b^d)
* Memory complexity: O(b^d)
* High memory requirements limit practical use for deep problems

Depth-First Search (DFS)

* Memory efficient: O(bm) where m is maximum depth
* Not complete in infinite spaces
* Not optimal - may find longer solutions
* Performance varies based on path distribution

Depth-Limited Search (DLS)

* DFS with predefined depth limit
* Memory requirements: O(bL) where L is depth limit
* Prevents infinite loops
* Misses solutions deeper than limit

Iterative Deepening (IDDFS)

* Combines BFS completeness with DFS memory efficiency
* Optimal for uniform-cost problems
* Time complexity: O(b^d) but with repeated work
* Memory complexity: O(bd)

Performance Tradeoffs

| Algorithm | Completeness | Optimality | Memory Usage | Speed |
| --- | --- | --- | --- | --- |
| BFS | Yes | Yes | Very High | Slow for deep solutions |
| DFS | No | No | Low | Fast if right path chosen |
| DLS | No | No | Low | Depends on limit choice |
| IDDFS | Yes | Yes | Low | Medium (repeats work) |

Selection Guidelines

* BFS: Use when solution is likely shallow and memory is available
* DFS: Use with limited memory when any solution is acceptable
* DLS: Choose when approximate solution depth is known
* IDDFS: Best general-purpose uninformed algorithm balancing memory and completeness

For Blocks World specifically, DLS performs efficiently when an appropriate depth limit is selected, avoiding the extensive memory requirements of BFS while preventing the infinite exploration possible with standard DFS.

**Conclusion:**

**Post Lab Objective Questions**

**1. Which is not a Goal-based agent?**

1. Inference
2. Search
3. Planning
4. Conclusion
5. Dynamic search.

**Answer: Conclusion**

**2. Which were built in such a way that humans had to supply the inputs and  
interpret the outputs?**

1. Agents
2. Sensor
3. AI System
4. Actuators

**Answer: AI System**

**Post Lab Subjective Questions**

**Post Lab Objective questions**

1. **Which search algorithm imposes a fixed depth limit on nodes?**
   1. Depth-limited search
   2. Depth-first search
   3. Iterative Deepening search
   4. Only (a) and (b)
   5. Only (a), (b) and (c).

**Answer: Depth-limited search**

1. **Optimality of BFS is**
   1. When all step costs are equal
   2. When all step costs are unequal
   3. When there is less number of nodes
   4. Both a & c

**Answer: When all step costs are equal**

1. What is a common application of Depth-First Search?
   1. Finding the shortest path
   2. Solving puzzles
   3. Web crawling
   4. Scheduling processes

**Answer: Solving puzzles**

**Post Lab Subjective Questions:**

1. **Provide a real-world example where BFS would be preferable over DFS, and another where DFS would be a better choice. Explain your reasoning.**

**Ans:**

BFS is preferable for finding the shortest path in an unweighted graph, such as navigating a city map to find the shortest route between two locations because BFS guarantees the shortest path.

DFS is better for solving puzzles like mazes or Sudoku, where exploring all possible configurations is required, as it dives deep into one branch before backtracking, efficiently exploring paths.

1. **If your search algorithm did not find a solution, what could be the possible reasons? How would adding a depth limit to DFS affect the outcome?**

**Ans:**

Reasons for failure: Infinite loops in cyclic graphs, insufficient depth limit, or unsolvable problems.

Adding a depth limit prevents infinite exploration but may miss deeper solutions if the limit is too low.

1. **If you modified BFS to use a priority queue instead of a regular queue, how would this change the search behavior? What kind of search algorithm would it resemble?**

**Ans:**

Modifying BFS with a priority queue transforms it into Uniform-Cost Search (UCS), which expands nodes with the lowest cumulative cost rather than by level, making it suitable for weighted graphs where costs vary between edges.