**Batch: Roll No.:**

**Experiment / assignment / tutorial No. 5**

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

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

|  |
| --- |
| **Title:** Implementation of informed search algorithm(Greedy Best First search/A\*) |

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Objective:** Examine the efficiency and performance of informed 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.

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

**New Concepts to be learned:**

Informed search.

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

**Informed Searching Technique**

Greedy Best-First Search:

* Chooses the path that appears closest to the goal using a heuristic.
* Quick but might not find the optimal path.

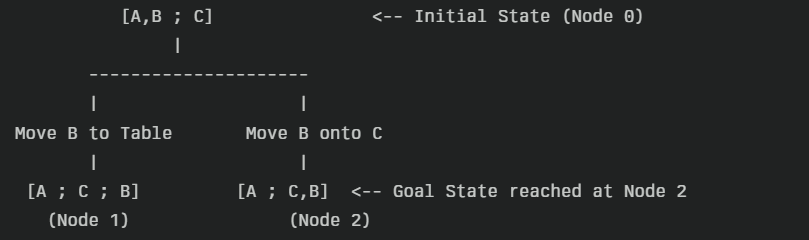
***A*\* Search:**

* Combines path cost and heuristic to find the best path.
* Ensures the optimal path if the heuristic is accurate.

**Problem Statement:**

In the Blocks World, you need to transform an initial configuration of blocks into a given goal configuration using legal moves. Each move consists of relocating only the top block from one stack to either the table (forming a new stack) or onto another stack, as long as that block is clear. The task is to find a sequence of moves that transforms the initial state into the goal state using the A\* search algorithm, where a heuristic guides the search to favor states that are closer to the goal.

**State-space tree :**

****

Node 0 (Initial): Stack 1 contains [A, B] and Stack 2 contains [C].

From Node 0, one legal move is moving the top block B from Stack 1.

Option 1: Moving B to the table yields a new state: [A] ; [C] ; [B] (Node 1). This state is not the goal because it creates an extra stack.

Option 2: Moving B onto block C (which is the top of Stack 2) creates the state [A] ; [C, B] (Node 2), which matches the goal.

**Solution with of chosen algorithm on the state-space tree:** *(attach solution solved on paper)*

Final A\* Solution

Based on the evaluated state-space, the A\* search identifies the following optimal solution:

Step 1: Move the top block B from the stack [A, B] onto block C in the stack [C].

This concise solution is achieved because the heuristic function (for example, counting misplaced blocks) recognizes that only one block is out of position in the initial state. The search algorithm thus directly expands the branch leading to the goal state through the move “B onto C.”

In a paper-solved representation, you would draw the above tree diagram to illustrate the explored nodes and clearly highlight the optimal solution path found by the A\* algorithm.

**Source code:**

import heapq

class BlocksWorld:

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

        """

        Initialize the 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 exactly matches the goal state.

        Here we assume the order of stacks and blocks is important.

        """

        return state == self.goal\_state

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

        """

        Generate all valid moves from the current state.

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

        is either another block (if moving onto it) or "Table" (to move the block to a new stack).

        """

        moves = []

        # Identify all blocks that can be moved (the top block of every stack)

        movable\_blocks = [stack[-1] for stack in state if stack]

        for block in movable\_blocks:

            # Moving to the table always forms a new stack.

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

            # Moving to any other movable block (which is on top of some stack)

            for target in movable\_blocks:

                if target != block:

                    moves.append((block, target))

        return moves

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

        """

        Apply a move (block, destination) to the state and return the new state.

        """

        block, destination = move

        # Create a deep copy of the state.

        new\_state = [list(stack) for stack in state]

        source\_stack\_idx = None

        # Locate the source stack from which 'block' is to be moved.

        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 the top

        # Remove the block from its original location.

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

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

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

        # Place the block according to the move.

        if destination == "Table":

            new\_state.append([block])

        else:

            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

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

        return new\_state

    def state\_to\_tuple(self, state):

        """

        Convert the state into a hashable tuple of tuples.

        """

        return tuple(tuple(stack) for stack in state)

    def heuristic(self, state):

        """

        A simple heuristic: count the number of blocks that are not in their goal position.

        The goal position of each block is defined by its location (stack index and position).

        """

        h = 0

        state\_pos = {}

        goal\_pos = {}

        for i, stack in enumerate(state):

            for j, block in enumerate(stack):

                state\_pos[block] = (i, j)

        for i, stack in enumerate(self.goal\_state):

            for j, block in enumerate(stack):

                goal\_pos[block] = (i, j)

        for block in goal\_pos:

            if block not in state\_pos or state\_pos[block] != goal\_pos[block]:

                h += 1

        return h

    def astar\_search(self):

        """

        Perform A\* search to find a sequence of moves from the initial state to the goal state.

        Each queue entry is a tuple (f, g, state, path) where 'f' is the total cost (g + h),

        'g' is the cost so far, and 'path' is the list of moves taken.

        """

        start = self.initial\_state

        start\_h = self.heuristic(start)

        queue = []

        heapq.heappush(queue, (start\_h, 0, start, []))

        visited = set()

        visited.add(self.state\_to\_tuple(start))

        while queue:

            f, g, state, path = heapq.heappop(queue)

            if self.is\_goal(state):

                return path

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

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

                if new\_state is not None:

                    state\_key = self.state\_to\_tuple(new\_state)

                    if state\_key not in visited:

                        visited.add(state\_key)

                        new\_path = path + [move]

                        new\_g = g + 1  # each move costs 1

                        new\_f = new\_g + self.heuristic(new\_state)

                        heapq.heappush(queue, (new\_f, new\_g, new\_state, new\_path))

        return None

def display\_solution(solution):

    if solution is None:

        print("No solution found.")

    else:

        print("Solution found with", len(solution), "move(s):")

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

            block, destination = move

            dest\_str = "the table" if destination == "Table" else f"block {destination}"

            print(f"Step {i}: Move block {block} to {dest\_str}")

def interactive\_mode():

    """

    Interactive mode to run the Blocks World A\* search.

    The user enters the initial and goal states as strings.

    Example format: A,B;C,D (two stacks: ['A', 'B'] and ['C', 'D'])

    """

    print("Blocks World A\* Search")

    init\_str = input("Enter initial state: ")

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

    def parse\_state(s):

        state = []

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

            if stack\_str:

                state.append(stack\_str.split(','))

        return state

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

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

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

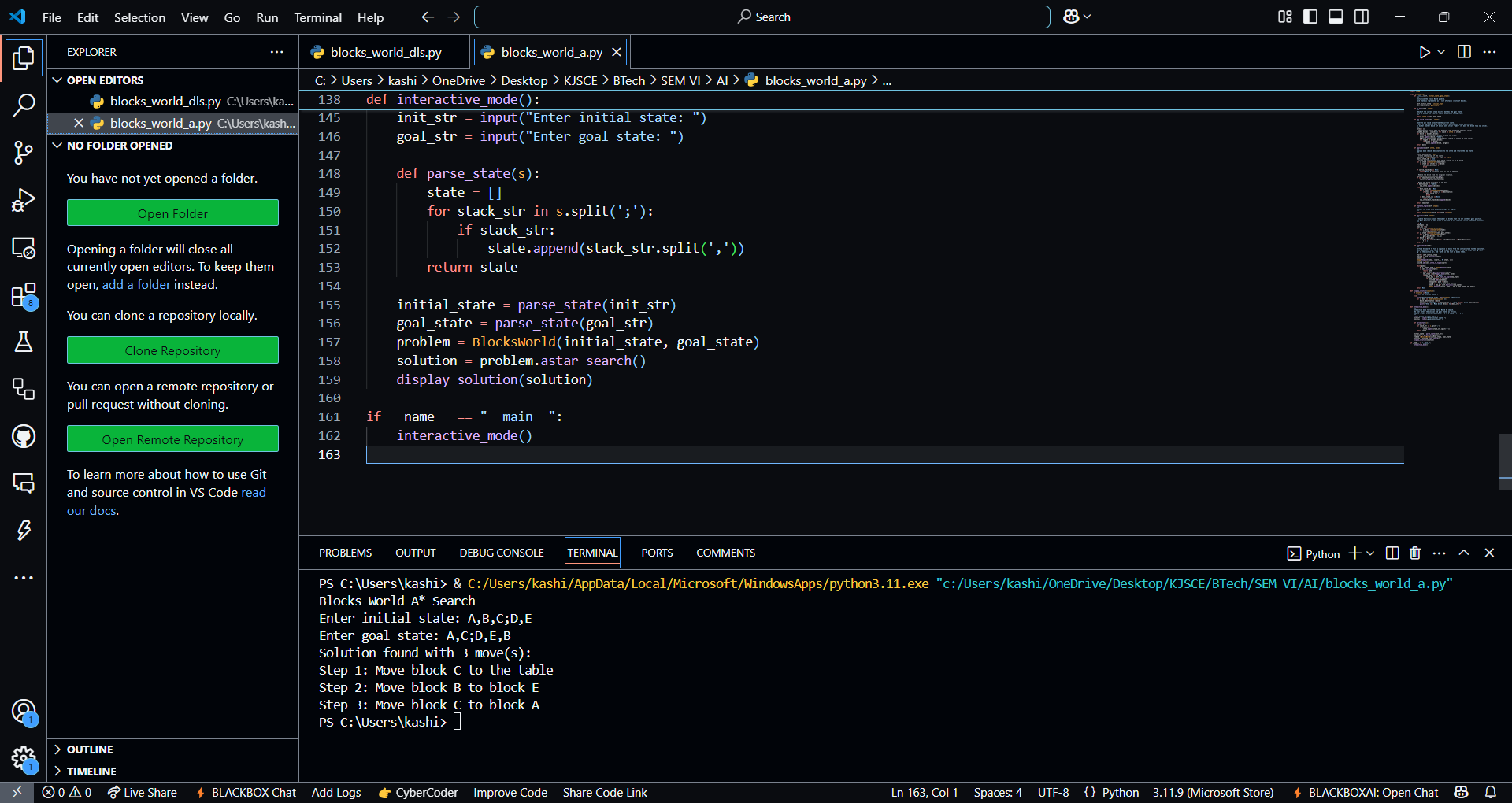
    solution = problem.astar\_search()

    display\_solution(solution)

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

    interactive\_mode()

**output screenshots:**



**Comparison of performance of Greedy and A\* Algorithm:**

Performance Comparison

* **Evaluation Function:**
  + **A\*:** Uses f(n)=g(n)+h(n)*f*(*n*)=*g*(*n*)+*h*(*n*), where g(n)*g*(*n*) is the cost so far and h(n)*h*(*n*) is the estimated cost to the goal. This dual component helps A\* to balance between the traveled path and future expectations
  + **Greedy Best-First Search:** Relies solely on h(n)*h*(*n*) to decide which node to expand next, which often makes it faster but less reliable in finding the optimal solution
* **Completeness and Optimality:**
  + **A\*:** If the heuristic is admissible (and consistent), A\* is both complete and optimal—it is guaranteed to find the best (shortest or least-cost) solution
  + **Greedy Best-First Search:** Generally not complete or optimal, as it may get trapped in local minima or overlook a better path in favor of rapidly promising nodes
* **Memory Consumption:**
  + **A\*:** Tends to use more memory because it stores all explored nodes to guarantee optimality
  + **Greedy Best-First Search:** Usually more memory-efficient as it often explores fewer nodes; however, this can come at the cost of solution quality

**Properties of A\* algorithm:**

Key Properties of A\* Algorithm

* **Admissible Heuristic:** An admissible h(n)*h*(*n*) ensures that A\* never overestimates the actual cost to reach the goal, thus guaranteeing optimality
* **Complete and Optimal:** With an admissible (and ideally consistent) heuristic, A\* will always find a solution if one exists and that solution is the optimal one
* **Balancing Cost and Estimate:** By combining g(n)*g*(*n*) and h(n)*h*(*n*), A\* strikes a balance between exploring the already traveled path and the estimated cost to finish, which often results in more efficient search paths
* **Memory Intensive:** Since A\* needs to keep track of all generated nodes to avoid re-expansion, its space complexity can be a limiting factor in very large search spaces
* **Guaranteed Termination:** In finite search spaces and with an admissible heuristic, A\* will eventually terminate, even though the worst-case time complexity can be exponential

In summary, while Greedy Best-First Search may be faster and use less memory due to its focus solely on the heuristic h(n)*h*(*n*), it sacrifices the guarantees of completeness and optimality that A\* provides through its combined cost function. This makes A\* the preferred choice for finding optimal paths when memory resources are available and ensuring that the best solution is found is critical.

**Conclusion:**

Heuristic search strategies like A\* and Greedy Best-First Search effectively reduce search effort by guiding exploration with heuristics, but their success depends on the accuracy and admissibility of the heuristic function.

**Post lab Objective questions**

1. **A heuristic is a way of trying**
2. To discover something or an idea embedded in a program
3. To search and measure how far a node in a search tree seems to be from a goal
4. To compare two nodes in a search tree to see if one is better than the other
5. Only (a) and (b)
6. Only (a), (b) and (c).

**Answer: Only (a), (b) and (c).**

1. **A\* algorithm is based on**
2. Breadth-First-Search
3. Depth-First –Search
4. Best-First-Search
5. Hill climbing.
6. Bulkworld Problem.

**Answer: Best-First-Search**

**3. What is a heuristic function?**

1. A function to solve mathematical problems
2. A function which takes parameters of type string and returns an integer value
3. A function whose return type is nothing
4. A function which returns an object
5. A function that maps from problem state descriptions to measures of desirability.

**Answer: A function that maps from problem state descriptions to measures of desirability.**

**Post Lab Subjective Questions:**

1. **How does the Greedy Best-First Search algorithm use a heuristic evaluation function?**

**Ans:**

Greedy Best-First Search uses the heuristic function h(n)*h*(*n*) to estimate how close a node is to the goal and selects for expansion the node with the lowest h(n)*h*(*n*). It ignores the path cost to reach that node, focusing solely on what appears most promising based on the heuristic value.

1. **Find a good heuristic function for following**
2. **Monkey and Banana problem**
3. **Travelling Salesman problem**

**Ans:**

Monkey and Banana Problem:

A good heuristic is the Manhattan (or Euclidean) distance between the monkey’s current position and the banana (or, if a box is required, the combined distance from the monkey to the box plus the distance from the box to the banana).

Travelling Salesman Problem:

One effective heuristic is to compute a Minimum Spanning Tree (MST) cost over the remaining unvisited cities and add the smallest cost edge from the current city to the MST. This gives a lower bound estimate for completing the tour.

1. **Define the heuristic search. Discuss benefits and short comings.**

**Ans:**

Heuristic search is a strategy that uses heuristic functions to guide the exploration of the search space by estimating the cost or distance to a goal from any given state.

Benefits:

It reduces search time by prioritizing promising nodes.

It can significantly cut down the number of nodes explored compared to uninformed search methods.

When the heuristic is admissible and consistent, it often yields optimal or near-optimal solutions.

Shortcomings:

An inaccurate or non-admissible heuristic may mislead the search, resulting in suboptimal solutions or failure to find a solution.

Design of an effective heuristic can be challenging, and over-reliance on heuristics might cause the algorithm to overlook viable alternatives.

These answers summarize the key concepts behind heuristic functions and the performance aspects of Greedy Best-First and A\* algorithms.