Project Report

United We Stand

A Brief Discussion of the Problem

The problem at hand involves controlling a set of micro-organisms on a rectangular grid, with the goal of merging all the micro-organisms into a single, larger organism. Each micro-organism occupies a single square on the grid and can move in four directions: North, South, East, and West. The grid is surrounded by harmful chemical deposits, which represent the boundaries beyond which the organisms cannot move.

The movement of each organism can result in three possible outcomes:

1. **Collision with Another Organism**: When two organisms occupy contiguous squares, they merge into a single, larger organism.
2. **Collision with an Obstacle**: The movement stops upon encountering an obstacle, and the organism remains in the square adjacent to the obstacle.
3. **Collision with the Chemical Boundary**: The organism is destroyed upon hitting the grid boundary.

The challenge is to design a controller that can issue movement instructions to the micro-organisms in such a way that all of them merge into a single organism while avoiding obstacles and boundaries. This problem is approached using various search algorithms, including Breadth-First Search (BFS), Depth-First Search (DFS), Iterative Deepening Search (IDS), Greedy Search with different heuristics, and A Search with admissible heuristics. The performance of these algorithms is compared in terms of runtime, the number of expanded nodes, memory utilization, and CPU utilization.

Search-tree Node ADT Implementation

The node class is the fundamental building block of a search tree. Instances of this class represent tree nodes, encapsulating important attributes such as the state, parent node, action, depth, and path cost. Here is a detailed discussion of the components and attributes of the node class:

1. **Attributes**:
    - **State**: This attribute holds the current state of the grid, including the positions of all organisms and obstacles. The state object is central to determining the configuration of the grid at any given node.
    - **Parent**: Each node has a reference to its parent node. This link to the parent node is essential for tracing back the path from the current node to the root node, thereby reconstructing the sequence of actions taken.
    - **Action**: The action that led to the creation of the current node from its parent. This helps in understanding the specific move that transitioned the system from the parent's state to the current state.
    - **Path Cost**: This represents the cumulative cost incurred to reach the current node from the root node. It is used in evaluating the desirability of paths, especially in cost-sensitive search algorithms.
    - **Depth**: The depth attribute indicates how many steps (actions) have been taken from the root node to the current node. It helps in limiting the search depth in algorithms like Iterative Deepening Search.

2. **Constructor**:
    - The node class has two constructors. The primary constructor initializes a node with a given state, parent node, action, path cost, and depth. The secondary constructor is a convenience constructor that initializes a node with only the state, defaulting the other attributes to null or zero.
3. **Getter Methods**:
    - **getState()**: Returns the current state of the node.
    - **getParent()**: Returns the parent node.
    - **getAction()**: Returns the action that led to the current node.
    - **getPathCost()**: Returns the cumulative path cost to reach the current node.
    - **getDepth()**: Returns the depth of the node in the search tree.
4. **Generating Successors**:
    - The getSuccessors() method generates all possible successor nodes from the current node. It iterates through each organism and each possible direction to create new states. Valid successor nodes are those where the resulting state is within bounds and does not collide with obstacles or other organisms.
5. **State Transition**:
    - The applyAction() method applies a given action to the current state to generate a new state. It ensures the new state is valid by checking boundaries and obstacles, and it calculates the cost of the action based on the distance moved and the size of the organism.
6. **Movement and Validation**:
    - The movePosition() and getNewPosition() methods handle the movement of organisms in the specified direction and validate the new positions against the grid's boundaries and obstacles.
7. **Utility Methods**:
    - **toString()**: Provides a string representation of the node's key properties, which is useful for debugging and logging.
    - **calculateDistance()**: Calculates the Manhattan distance between two positions, used for cost calculation in state transitions.

Search Problem ADT Implementation

The GenericSearch class is an abstract framework for implementing search algorithms, providing a standardized structure for defining and executing search problems. It includes a single abstract method, search(Node initialState),which initiates the search from a given starting node and returns a node representing the solution or final state. This class establishes a common interface for different search strategies, allowing subclasses to specify their own search methodologies while adhering to the overall framework. By defining this foundational method, GenericSearch facilitates the development of various search algorithms, ensuring consistency and flexibility in exploring and solving search problems.

United-We-Stand Problem ADT Implementation

The UnitedWeStandSearch class implements various search strategies for solving the "United We Stand" problem, where the goal is to ensure that a grid-based problem eventually has only one

organism left. The core of the implementation revolves around the `Node` and `State` classes, which represent the search nodes and states respectively. The `evaluationFunction` method combines path cost and heuristic evaluations to prioritize nodes in the search. Heuristic functions such as heuristic1and heuristic2 are used to estimate the cost to the goal. heuristic1is a straightforward heuristic that counts the number of organisms, while heuristic2 calculates the sum of the minimum distances from each organism to the nearest obstacle, providing a more nuanced estimation. The applyAction method simulates the result of moving an organism in the specified direction, modifying the state accordingly. The class also features methods like expandNode to generate successor nodes and goalTest to determine if the search goal is met.

Main Functions Implementation

The UnitedWeStandSearch class extends GenericSearch and provides various functionalities for solving search problems with different strategies. The evaluationFunction() method calculates the evaluation score for a node by summing its path cost and a heuristic value, which in this case, is computed by the heuristic1() method. This method returns the number of organisms in the node's state. Two other heuristic methods, heuristic1 and heuristic2(), are defined to guide the search. While heuristic() simply counts the organisms, heuristic2() computes the total minimum distance from each organism to the nearest obstacle, helping to estimate the cost to reach the goal state. The goalTest method checks if a node's state meets the goal condition, which is having exactly one organism left. The stepCost() method provides a cost estimate for moving from one node to another, with a default value of 1. The applyAction() method generates a new state by applying a specified action to the current state, moving an organism in a given direction (north, south, east, or west). It also handles edge cases, such as invalid organism indices.
The getPossibleActions method generates all valid actions for the current state, considering all possible moves for each organism in all four directions. The genGrid() method creates a random grid configuration with organisms and obstacles, producing a string representation of the grid's dimensions and contents.
The containsState() helper method checks if a state is present in a priority queue. The solve() method initializes the search with the given grid configuration and strategy, choosing the appropriate search algorithm (such as breadth-first, depth-first, or A search) based on the strategy specified. It then constructs and returns the solution path, including actions, path cost, and the number of expanded nodes. If visualized, the display() method prints the grid's state at each step of the solution path.
Additionally, the totalCost() and totalCost2() methods calculate the total cost of a node by combining path cost and heuristic values. The parseGrid method converts a grid string into a State object by extracting dimensions, organisms, and obstacles. Lastly, the constructPath method traces back from a goal node to the initial node, constructing the solution path in reverse.

Search Algorithms Implementation

The UnitedWeStandSearch class implements various search algorithms to tackle problems involving node exploration. Here's a breakdown of how each algorithm is implemented:

1. Breadth-First Search (BFS):
   The BFS algorithm uses a PriorityQueue to explore nodes, initialized with the initialState. Nodes are expanded by dequeuing from the frontier, and their neighbors are added to the frontier if they haven't been explored. The goalTest method is used to check if the current node meets the goal condition. If the goal is reached, the node is returned; otherwise, the search continues until the frontier is empty.

2. Depth-First Search (DFS):
   DFS utilizes a Stack for frontier management, pushing the initialState onto the stack. Nodes are expanded by popping from the stack, and their successors are added to the stack if they haven't

been explored. Similar to BFS, the goalTest is applied to determine if the goal has been reached. The process continues until the stack is empty or the goal state is found.

3. Uniform Cost Search (UCS):
UCS employs a PriorityQueue where nodes are prioritized based on their path cost. Nodes are expanded by dequeuing the node with the lowest path cost. Successors are added to the frontier if they haven't been explored or if a cheaper path to the state is found. The path cost for each successor is updated accordingly, and the process continues until the goal is reached or the frontier is empty.

4. Iterative Deepening Search (IDS):
IDS starts with a depth limit of 0 and increases the limit incrementally. For each depth limit, a Stack is used to explore nodes. Nodes are expanded within the current depth limit, and their successors are pushed onto the stack if they do not exceed the current depth. The search continues until the goal is found or if the number of expanded nodes at the current depth matches the number from the previous depth, indicating no progress.

5. Greedy Search 1:
This algorithm initializes a PriorityQueue based on a heuristic function (heuristic1). Nodes are prioritized by their heuristic value, and successors are added to the frontier if they haven't been explored. The goal state is checked after each node is dequeued. The process continues until the goal is reached or the frontier is empty.

6. Greedy Search 2:
Similar to Greedy Search with Heuristic 1, this version uses heuristic2 to prioritize nodes in the PriorityQueue. The heuristic value helps guide the search towards the goal. Successors are explored and added to the frontier if they haven't been visited. The search proceeds until the goal state is found or the frontier is empty.

7. AS1:
A Search uses a PriorityQueue where nodes are prioritized based on the sum of the path cost and heuristic value (totalCost). The cost to reach each state is tracked in a costMap, and successors are added to the frontier with updated path costs. If a cheaper path to a state is found, the cost is updated. The search continues until the goal is reached.

8. AS2:
This variant of A Search uses heuristic2 for the priority calculation. The process is similar to A Search with Heuristic 1, but with a different heuristic function influencing node prioritization. The costMap2 is used to track path costs, and nodes are expanded based on their total estimated cost. The search continues until the goal state is reached or the frontier is empty.

In the implementation of search algorithms, heuristic functions play a crucial role, especially in algorithms like Greedy Search and A Search. Here’s a discussion of the heuristic functions used and their admissibility in the context of A Search:

Heuristic Functions

1. Heuristic Function 1 (heuristic1()):
- Purpose: This heuristic function estimates the cost to reach the goal from a given node.
- Characteristics: The exact nature of heuristic1 is not specified in the provided code, but it generally aims to provide a lower bound on the remaining cost to the goal.
- Role in Greedy Search: In Greedy Search using heuristic1, the algorithm prioritizes nodes based on this heuristic value. This helps guide the search towards nodes that appear closer to the goal, based on the heuristic estimate.
- Role in A Search: In A Search, heuristic1 contributes to calculating the total estimated cost (totalCost), which is the sum of the path cost from the start node and the heuristic estimate. This total cost helps A search prioritize nodes more effectively, balancing between exploring nodes with lower path costs and those with promising heuristic estimates.

2. Heuristic Function 2 (heuristic2()):
- Purpose: This heuristic function is an alternative estimate of the cost to reach the goal.
- Characteristics: As with heuristic1, the exact nature of heuristic2 is not specified, but it is designed to provide a different perspective on the cost to reach the goal.
- Role in Greedy Search: Similar to heuristic1, heuristic2 is used to guide the search by prioritizing nodes with lower heuristic values.
- Role in A Search: In A Search, heuristic2 is used to compute the total estimated cost (totalCost2). It helps prioritize nodes in a way that balances the path cost and heuristic estimate.

Admissibility of Heuristic Functions in A Search

In the context of AS Search, a heuristic function is considered admissible if it never overestimates the true cost to reach the goal. Admissibility ensures that A Search is both complete (it will find a solution if one exists) and optimal (it will find the least-cost solution). Here's a discussion on the admissibility of the heuristic functions used:

1. Admissibility of heuristic1() :
- Definition: For heuristic1 to be admissible, it must satisfy the condition that for any node, the heuristic value should be less than or equal to the true cost from that node to the goal.
- Argument for Admissibility: If heuristic1 is designed such that it consistently provides an underestimate of the true cost, it is admissible. This means that heuristic1 will never provide a heuristic value that exceeds the actual cost to reach the goal from any node. If heuristic1 meets this condition, then A Search using heuristic1 will be both complete and optimal.

2. Admissibility of heuristic2() :
- Definition: Similarly, heuristic2 must also ensure that the estimated cost it provides is always less than or equal to the actual cost from the node to the goal.
- Argument for Admissibility: If heuristic2 is carefully designed to provide a lower bound on the true cost, it will be admissible. By ensuring that heuristic2 never overestimates the actual cost, A Search using heuristic2 will also be complete and optimal.

Performance Summary

BFS:

Test a1:
CPU Usage from Activity Monitor: 19.81%
Time Elapsed: 0.003 seconds

Test a2:
CPU Usage from Activity Monitor: 17.25%
Elapsed time: 18.382 seconds

DFS:

Test b1:
CPU Usage from Activity Monitor: 23.75%
Time Elapsed: 0.002 seconds

Test b2:
CPU Usage from Activity Monitor: 20.12%
Elapsed time: 0.017 seconds

UC:

Test c1:
CPU Usage from Activity Monitor: 16.89%
Time Elapsed: 0.002 seconds

Test c2:
CPU Usage from Activity Monitor: 31.85%
Elapsed time:org.junit.runners.model.TestTimedOutException: test timed out after 120000 milliseconds

---

IDS:

Test d1:
CPU Usage from Activity Monitor: 10.33%
Time Elapsed: 0.001 seconds

Test d2:
CPU Usage from Activity Monitor: 17.73%
Elapsed time: 29.896 seconds

GR1:

Test e1:
CPU Usage from Activity Monitor: 10.68%
Time Elapsed: 0.002 seconds

Test e2:
CPU Usage from Activity Monitor: 22.80%
Elapsed time: 0.006 seconds

GR2:

Test f1:
CPU Usage from Activity Monitor: 10.09%
Time Elapsed: 0.002 seconds

Test f2:
CPU Usage from Activity Monitor: 6.55%
Elapsed time: 2.204 seconds

AS1:

Test g1:
CPU Usage from Activity Monitor: 8.12%
Time Elapsed: 0.002 seconds

Test g2:
CPU Usage from Activity Monitor: 8.25%
Elapsed time: 31.607 seconds

AS2:

Test h1:
CPU Usage from Activity Monitor: 16.04%
Time Elapsed: 0.001 seconds

Test h2:
CPU Usage from Activity Monitor: 7.58%
Elapsed time: 58.59 seconds

Performance Comparison and Discussion

In evaluating the performance of various search algorithms, we observe notable differences in CPU usage and elapsed time. For Breadth-First Search (BFS), the CPU usage ranges from 17.25% to 19.81%, with elapsed times varying from a quick 0.003 seconds in simpler cases to a more substantial 18.382 seconds in complex scenarios. Depth-First Search (DFS) shows slightly higher CPU usage, ranging from 20.12% to 23.75%, with elapsed times from 0.002 to 0.017 seconds. Uniform Cost Search (UC) displays lower CPU usage in simple cases (16.89%) but experiences a dramatic increase to 31.85% and a timeout in more challenging tests, indicating scalability issues. Iterative Deepening Search (IDS) has relatively low CPU usage (10.33% to 17.73%) but exhibits longer elapsed times, with up to 29.896 seconds in more complex cases. Greedy Searches (GR1 and GR2) have efficient CPU usage, from 6.55% to 22.80%, with elapsed times varying from 0.002 seconds to 2.204 seconds. A* Search (AS1 and AS2) also maintains low CPU usage (7.58% to 16.04%) but encounters significant delays, with elapsed times reaching up to 58.59 seconds. Overall, while BFS, DFS, and IDS demonstrate faster execution times, algorithms like A* and Greedy Search offer better CPU efficiency but may suffer from longer execution times, underscoring the need to balance CPU usage with elapsed time depending on the problem's complexity.