ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

LAB 1: SEARCH IN GRAPH

Lớp: 20VP

20126041 – Nguyễn Huỳnh Mẫn

Môn học: Cơ sở trí tuệ nhân tạo

Thành phố Hồ Chí Minh – 2023

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

LAB 1: SEARCH IN GRAPH

| Giáo viên hướng dẫn |

Thầy Lê Hoài Bắc

Thầy Nguyễn Bảo Long

Thầy Nguyễn Ngọc Đức

Môn học: Cơ sở trí tuệ nhân tạo

Thành phố Hồ Chí Minh – 2023

TABLE OF CONTENTS

Search Problem:

- A search problem is a fundamental concept in computer science and artificial intelligence. It involves finding a solution in a search space, typically a graph or a tree, starting from an initial state and proceeding through a sequence of actions to reach a goal state while satisfying certain constraints. The key elements of a search problem include:

1. Initial State: The state from which the search begins.
2. Successor Function: A function that generates the set of possible actions or states that can be reached from the current state.
3. Goal Test: A test to determine whether a given state is a goal state or not.
4. Path Cost Function: A function that assigns a cost to each action or state transition.

General Pseudo Code for Search Problem:

Here's a general pseudo code to solve a search problem:

```
function search_problem(initial_state):
  // Initialize data structures
  open_list = [initial_state]
  closed_set = []

  while open_list is not empty:
    // Select a state from the open list
    current_state = select_state(open_list)

    // Check if the current state is a goal state
    if is_goal_state(current_state):
      return reconstruct_solution(current_state)

    // Generate successor states
    successors = generate_successors(current_state)

    for successor in successors:
      if successor not in closed_set:
        if successor not in open_list or cost(current_state) + cost_to_successor < cost(successor):
          // Update cost and path to successor
          set_cost_and_path(successor, current_state, cost(current_state) + cost_to_successor)

          if successor not in open_list:
            // Add successor to open list
            add_to_open_list(successor)

    // Move current state to closed set
    add_to_closed_set(current_state)
```

```
// No solution found
return "No solution"
```

Distinguishing Uninformed Search and Informed Search:

- Uninformed Search: Uninformed search algorithms, such as Depth-First Search (DFS) and Breadth-First Search (BFS), do not use any domain-specific knowledge to guide the search. They explore the search space systematically without considering the cost or utility of the states. These algorithms are typically used when there is no information about the problem domain.
- Informed Search: Informed search algorithms, such as Uniform Cost Search (UCS) and A* Search, use domain-specific knowledge to guide the search. They consider the cost or heuristic information associated with states to make more informed decisions about which states to explore. Informed search algorithms are often more efficient and effective in finding solutions compared to uninformed search algorithms.

Algorithm

DFS algorithm

- Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.
- The DSF algorithm follows as:
  - We will start by putting any one of the graph's vertex on top of the stack.
  - After that take the top item of the stack and add it to the visited list of the vertex.
  - Next, create a list of that adjacent node of the vertex. Add the ones which aren't in the visited list of vertexes to the top of the stack.
  - Lastly, keep repeating steps 2 and 3 until the stack is empty.
- Completeness: DFS is complete if the search tree is finite, meaning for a given finite search tree, DFS will come up with a solution if it exists.
- Optimality: DFS is not optimal, meaning the number of steps in reaching the solution, or the cost spent in reaching it is high.
- Time complexity: Equivalent to the number of nodes traversed in DFS. T(n) = 1 + n^2 + n^3 + … + n^d = O(n^d)
- Space complexity: Equivalent to how large can the fringe get. S(n) = O(n x d)
  - d = the depth of the search tree = the number of levels of the search tree.
  - n^i = number of nodes in level i.

Pseudo code

- A recursive implementation of DFS

```
procedure DFS(G, v) is
  label v as discovered
  for all directed edges from v to w that are in G.adjacentEdges(v) do
    if vertex w is not labeled as discovered then
      recursively call DFS(G, w)
```

- A non-recursive implementation of DFS with worst-case space complexity O(|E|), with the possibility of duplicate vertices on the stack

```
procedure DFS_iterative(G, v) is
  let S be a stack
  S.push(v)
  while S is not empty do
    v = S.pop()
    if v is not labeled as discovered then
      label v as discovered
      for all edges from v to w in G.adjacentEdges(v) do
        S.push(w)
```

BFS algorithm

- Breadth-first search (BFS) is an algorithm for searching a tree data structure for a node that satisfies a given property. It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level.
- Breadth-First Traversal (or Search) for a graph is similar to the Breadth-First Traversal of a tree.
- The only catch here is, that, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we divide the vertices into two categories:
  - Visited and
  - Not visited.
- A boolean visited array is used to mark the visited vertices. For simplicity, it is assumed that all vertices are reachable from the starting vertex. BFS uses a queue data structure for traversal.
- The steps of the algorithm work as follow:
  - Start by putting any one of the graph's vertices at the back of the queue.
  - Now take the front item of the queue and add it to the visited list.
  - Create a list of that vertex's adjacent nodes. Add those which are not within the visited list to the rear of the queue.
  - Keep continuing steps two and three till the queue is empty.
- Completeness: BFS is complete, meaning for a given search tree, BFS will come up with a solution if it exists.

- Optimality: BFS is optimal as long as the costs of all edges are equal.
- Time complexity: Equivalent to the number of nodes traversed in BFS until the shallowest solution. T(n) = 1 + n^2 + n^3 + … + n^s = O(n^s)
- Space complexity: Equivalent to how large can the fringe get. S(n) = O(n^s)
  - s = the depth of the shallowest solution.
  - n^i = number of nodes in level i.

Pseudo code

- Input: A graph G and a starting vertex root of G
- Output: Goal state. The parent links trace the shortest path back to root

```
procedure BFS(G, root) is
    let Q be a queue
    label root as explored
    Q.enqueue(root)
    while Q is not empty do
      v := Q.dequeue()
      if v is the goal then
         return v
      for all edges from v to w in G.adjacentEdges(v) do
         if w is not labeled as explored then
            label w as explored
            w.parent := v
            Q.enqueue(w)
```

UCS algorithm

- In computer science, uniform-cost search (also known as minimum cost search or uniform cost search, is a variant of Dijikstra's algorithm, abbreviated in English as UCS) is a tree traversal method used for browsing or searching. a tree, tree structure, or weighted graph (cost). The search starts at the root node. The search continues by traversing the next nodes with the lowest weight or cost from the root node. Nodes continue to be traversed until the desired destination node is reached.
- Completeness: Complete (with finite path costs).
- Optimality: Optimal (with finite path costs).
- Time Complexity: O(m ^ (1+floor(l/e)))
  - where:
  - m is the maximum number of neighbors a node has
  - l is the length of the shortest path to the goal state
  - e is the least cost of an edge
- Space complexity: O(m ^ (1+floor(l/e)))
  - where:

- m is the maximum number of neighbors a node has
- l is the length of the shortest path to the goal state
- e is the least cost of an edge

Pseudo code

```
procedure uniform_cost_search(start) is
  node ← start
  frontier ← priority queue containing node only
  expanded ← empty set
  do
    if frontier is empty then
      return failure
    node ← frontier.pop()
    if node is a goal state then
      return solution(node)
    expanded.add(node)
    for each of node's neighbors n do
      if n is not in expanded and not in frontier then
        frontier.add(n)
      else if n is in frontier with higher cost
        replace existing node with n
```

Astar algorithm

- A* is an informed search algorithm, or a best-first search, meaning that it is formulated in terms of weighted graphs: starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (least distance travelled, shortest time, etc.) #### Why A* Search Algorithm?
- Informally speaking, A* Search algorithms, unlike other traversal techniques, it has “brains”. What it means is that it is really a smart algorithm which separates it from the other conventional algorithms. This fact is cleared in detail in below sections. And it is also worth mentioning that many games and web-based maps use this algorithm to find the shortest path very efficiently (approximation).

Pseudo code

```
// A* Search Algorithm
1.  Initialize the open list
2.  Initialize the closed list
   put the starting node on the open
   list (you can leave its f at zero)

3.  while the open list is not empty
   a) find the node with the least f on
     the open list, call it "q"

   b) pop q off the open list

   c) generate q's 8 successors and set their
     parents to q
```

d) for each successor
i) if successor is the goal, stop search

ii) else, compute both g and h for successor
successor.g = q.g + distance between
successor and q
successor.h = distance from goal to
successor (This can be done using many
ways, we will discuss three heuristics-
Manhattan, Diagonal and Euclidean
Heuristics)

successor.f = successor.g + successor.h

iii) if a node with the same position as
successor is in the OPEN list which has a
lower f than successor, skip this successor

iV) if a node with the same position as
successor is in the CLOSED list which has
a lower f than successor, skip this successor
otherwise, add the node to the open list
end (for loop)

e) push q on the closed list
end (while loop)

Heuristics

We can calculate g but how to calculate h ? We can do things.

A) Either calculate the exact value of h (which is certainly time consuming).

OR

B) Approximate the value of h using some heuristics (less time consuming). We will discuss both of the methods.

Exact Heuristics

We can find exact values of h, but that is generally very time consuming. Below are some of the methods to calculate the exact value of h. 1) Pre-compute the distance between each pair of cells before running the A* Search Algorithm. 2) If there are no blocked cells/obstacles then we can just find the exact value of h without any pre-computation using the distance formula/Euclidean Distance

Approximation Heuristics –

There are generally three approximation heuristics to calculate h –

1) Manhattan Distance

- It is nothing but the sum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively, i.e.,

```
h = abs (current_cell.x – goal.x) +
    abs (current_cell.y – goal.y)
```

- When to use this heuristic? – When we are allowed to move only in four directions only (right, left, top, bottom)

2) Diagonal Distance

- It is nothing but the maximum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively, i.e.,

```
dx = abs(current_cell.x – goal.x)
dy = abs(current_cell.y – goal.y)

h = D * (dx + dy) + (D2 - 2 * D) * min(dx, dy)
```

where D is length of each node(usually = 1) and D2 is diagonal distance between each node (usually = sqrt(2)).

- When to use this heuristic? – When we are allowed to move in eight directions only (similar to a move of a King in Chess)

3) Euclidean Distance

- As it is clear from its name, it is nothing but the distance between the current cell and the goal cell using the distance formula

```
h = sqrt ( (current_cell.x – goal.x)2 +
       (current_cell.y – goal.y)2 )
```

- When to use this heuristic? – When we are allowed to move in any directions.
- Completeness: Complete (with finite path costs).
- Optimality: Optimal (with an admissible heuristic).
- Time Complexity: The time complexity of A* depends on the quality of the heuristic function. In a worst-case, the algorithm can be O(b^d), where b is the branching factor – the average number of edges from each node, and d is the number of nodes on the resulting path.
- Space Complexity: The space complexity of standard A* is always O(b^d), since we need to track every node in the graph at all times, even ones that we've never visited and are never going to. There are optimizations around this that can be made by only adding nodes to our algorithm as they become relevant, or by forgetting nodes as they become less relevant. Still, these all have potential impacts on the overall output.
- Limitations: Although being the best path finding algorithm around, A* Search Algorithm doesn't produce the shortest path always, as it relies heavily on heuristics / approximations to calculate – h

Comparison

UCS, Greedy, AStar

- Uniform Cost Search (UCS), Greedy Search, and A* Search are graph search algorithms used in various pathfinding and optimization problems. Let's compare them based on their characteristics, use cases, and performance in the context of graph algorithms:

Uniform Cost Search (UCS):

- **Characteristics**:
  - UCS is an uninformed search algorithm that explores the search space based on the actual path cost from the start node.
  - It expands nodes with the lowest path cost first, effectively exploring the least expensive paths.
- **Use cases**:
  - UCS is suitable for finding the shortest path from a source node to a destination node in weighted graphs.
  - It is used when you want to find the path with the minimum cost.
- **Performance**:
  - Completeness: UCS is complete if path costs are finite, and it will find the optimal solution in terms of path cost.
  - Optimality: UCS is optimal for finding the path with the lowest cost, provided that path costs are non-negative.
  - Time Complexity: O(m ^ (1+floor(l/e)))
    - where:
    - m is the maximum number of neighbors a node has
    - l is the length of the shortest path to the goal state
    - e is the least cost of an edge
  - Space complexity: O(m ^ (1+floor(l/e)))
    - where:
    - m is the maximum number of neighbors a node has
    - l is the length of the shortest path to the goal state
    - e is the least cost of an edge

Greedy Search:

- **Characteristics**:
  - Greedy Search is an informed search algorithm that selects nodes based on a heuristic function that estimates how close a node is to the goal.
  - It prioritizes exploration of nodes that appear closest to the goal based on the heuristic.

  - No backtracking! No reevaluating choices that the algorithm committed to earlier.
- **Use cases**:
  - Greedy Search is suitable when you have a heuristic that provides a good estimate of node proximity to the goal.
  - It is often used when you prioritize reaching the goal quickly without concern for the overall path cost.
- **Performance**:
  - Completeness: Greedy Search is not guaranteed to be complete. It may get stuck in loops and fail to find a solution.
  - Optimality: Greedy Search is not guaranteed to be optimal. It may find a suboptimal path in terms of cost.
  - Time Complexity: assuming the intervals are already sorted by finish times, time complexity is O(n). Otherwise, it will be O(n log n).
  - Space Complexity: O(1).

A* Search:

- **Characteristics**:
  - A* Search is an informed search algorithm that combines path cost with a heuristic function to guide exploration.
  - It selects nodes based on a combination of the cost to reach the node and an estimate of the cost to reach the goal from the node.
- **Use cases**:
  - A* Search is versatile and widely used in various applications where both path cost and proximity to the goal matter.
  - It is appropriate when you have a heuristic that provides a good estimate, and you want to find an optimal solution.
- **Performance**:
  - Completeness: A* is complete if the branching factor is finite, and the heuristic is admissible.
  - Optimality: A* is optimal if the heuristic is admissible, meaning it never overestimates the cost to reach the goal.
  - Time Complexity: assuming the intervals are already sorted by finish times, time complexity is O(n). Otherwise, it will be O(n log n).
  - Space Complexity: O(1).
- **BONUS**:
  - Relation (Similarity and Differences) with other algorithms- Dijkstra is a special case of A* Search Algorithm, where h = 0 for all nodes.

Conclusion

- UCS (Uniform Cost Search), Greedy Search, and A* Search are all algorithms for graph search. They have distinct characteristics, completeness, optimality, and use cases. Let's compare them:

- **Completeness**:
  - UCS: UCS is complete, meaning it is guaranteed to find a solution if one exists in a finite graph. It explores nodes in order of increasing cost and will eventually reach the goal.
  - Greedy Search: Greedy search is not complete. It can get stuck in loops or miss the optimal solution due to its myopic focus on the heuristic.
  - A* Search: A* search is complete, provided that the heuristic used is admissible (never overestimates the true cost) and the graph is finite. It is guaranteed to find the optimal solution.
- **Optimality**:
  - UCS: UCS is optimal. It guarantees that it will find the shortest path if it exists.
  - Greedy Search: Greedy search is not optimal. It may find a solution, but it is not guaranteed to be the shortest path.
  - A* Search: A* search is optimal if the heuristic used is admissible. It will find the shortest path.
- **Time Complexity**:
  - UCS: UCS's time complexity depends on the branching factor, the depth of the shallowest goal, and the cost of the path to the shallowest goal. In the worst case, it can be exponential.
  - Greedy Search: Greedy search can have a lower time complexity than UCS, but it may not find the optimal solution.
  - A* Search: A* search has a time complexity similar to UCS, but it benefits from the use of heuristics, which can significantly reduce the search space.
- **Space Complexity**:
  - UCS: UCS's space complexity is O(b^d) in the worst case, where b is the branching factor, and d is the depth of the shallowest goal.
  - Greedy Search: Greedy search's space complexity is generally lower than UCS because it doesn't store as much information.
  - A* Search: A* search's space complexity is similar to UCS but can be more efficient due to pruning of nodes based on heuristics.
- **Characteristics**:
  - UCS: UCS explores nodes in order of increasing path cost, focusing on the least costly paths first.
  - Greedy Search: Greedy search is greedy and selects nodes based on a heuristic, making it myopic and biased toward nodes that seem closer to the goal.

  - A* Search: A* search combines the advantages of UCS and Greedy Search. It explores nodes based on their estimated total cost, balancing the path cost and the heuristic.
- **Use Cases**:
  - UCS: UCS is suitable for scenarios where you want to find the shortest path with varying edge costs. It is often used in route planning and navigation.
  - Greedy Search: Greedy search is useful when you need a quick solution that is "good enough" but not necessarily optimal. It is used in heuristic-based problems like the traveling salesman problem.
  - A* Search: A* search is widely applicable when you want an optimal solution while taking advantage of heuristics. It is used in various applications, including game AI, robotics, and pathfinding.
- In summary, UCS is the most reliable for finding the optimal path, but it can be slow. Greedy Search is faster but may not find the best solution. A* Search combines the benefits of both UCS and Greedy Search, offering a good balance between optimality and efficiency when an admissible heuristic is used. The choice of algorithm depends on the specific requirements of the problem.

UCS, Dijkstra

Dijkstra's Algorithm:

- **Characteristics**:
  - Dijkstra's Algorithm is a specialized shortest-path algorithm designed to find the shortest path from a single source node to all other nodes in a weighted graph.
  - It explores the search space by considering the actual path cost.
- **Use cases**:
  - Dijkstra's Algorithm is specifically used for finding the shortest path in graphs with non-negative edge weights.
  - It is suitable for problems like finding the shortest route in a road network or determining the minimum distance to all nodes in a network.
- **Performance**:
  - Completeness: Dijkstra's Algorithm is complete, and it will find the shortest path to all reachable nodes.
  - Optimality: Dijkstra's Algorithm is optimal for finding the shortest path with non-negative edge weights.
  - Time Complexity: O(E + |V|log|V|)
    - where: V is the number of nodes or vertices, E is the number of neighbors or edges.
  - Space Complexity: O(2*V).
    - where: V is the number of nodes or vertices.

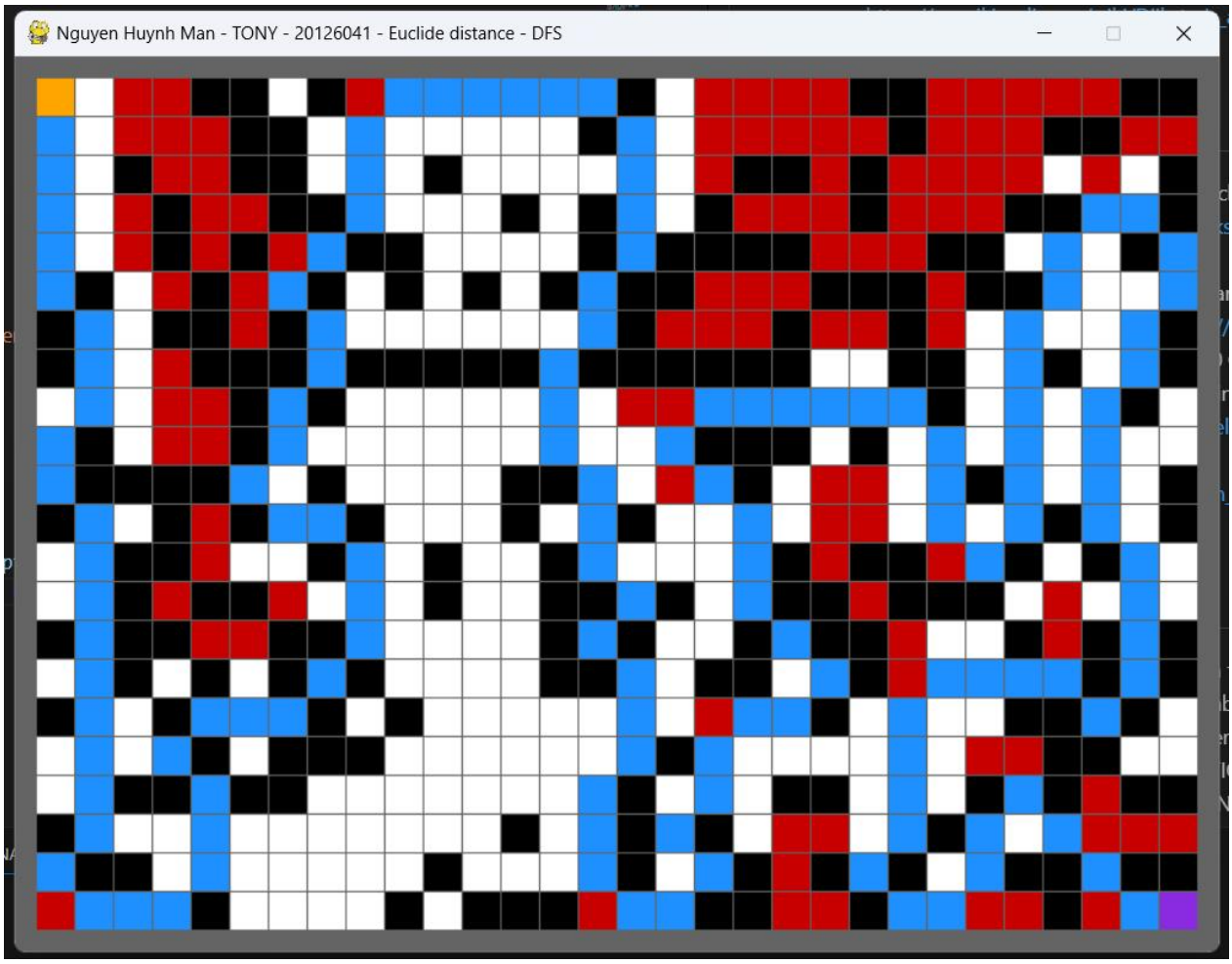
Conclusion

- **Completeness**:

  - UCS: UCS is complete, meaning it will find a solution if one exists in a finite graph. It explores all reachable nodes in increasing order of cost, so it is guaranteed to find the shortest path if it exists.
  - Dijkstra: Dijkstra's algorithm is also complete when applied to finite graphs with non-negative edge weights. It explores nodes in increasing order of distance from the start node.
- **Optimality**:
  - UCS: UCS is optimal. It guarantees that it will find the shortest path from the start node to all other reachable nodes.
  - Dijkstra: Dijkstra's algorithm is also optimal, provided that edge weights are non-negative. It finds the shortest path to all nodes from the start node.
- **Time Complexity**:
  - UCS: UCS's time complexity depends on the branching factor, the depth of the shallowest goal, and the cost of the path to the shallowest goal. In the worst case, it can be exponential.
  - Dijkstra: Dijkstra's time complexity is O((V + E) * log(V)) when using a priority queue, where V is the number of nodes and E is the number of edges. It can be slower if negative edge weights are present.
- **Space Complexity**:
  - UCS: UCS's space complexity is O(b^d) in the worst case, where b is the branching factor, and d is the depth of the shallowest goal. It can be quite high.
  - Dijkstra: Dijkstra's space complexity is O(V), where V is the number of nodes. It stores distances and nodes in the priority queue.
- **Characteristics**:
  - UCS: UCS explores nodes in order of increasing path cost, starting from the source node. It is blind to the actual distance to the goal but finds the optimal solution.
  - Dijkstra: Dijkstra's algorithm explores nodes based on their current distances from the source node. It is also optimal but requires non-negative edge weights.
- **Use Cases**:
  - UCS: UCS is useful in scenarios where you want to find the shortest path from a source node to a destination node with unknown or varying edge costs. It can be applied in route planning, maze solving, and artificial intelligence.
  - Dijkstra: Dijkstra's algorithm is commonly used for finding the shortest path in scenarios where edge weights are non-negative, such as network routing, GPS navigation, and road maps. It is widely used in real-world applications where non-negative distances between locations are common.
- In summary, both UCS and Dijkstra's algorithm are effective at finding optimal paths in graphs. UCS is more flexible in terms of edge weights but can be less efficient in terms of time and space complexity. Dijkstra's algorithm, on the other hand, is well-suited for scenarios with non-negative edge weights and is widely used in practical applications.

Implement

DFS algorithm



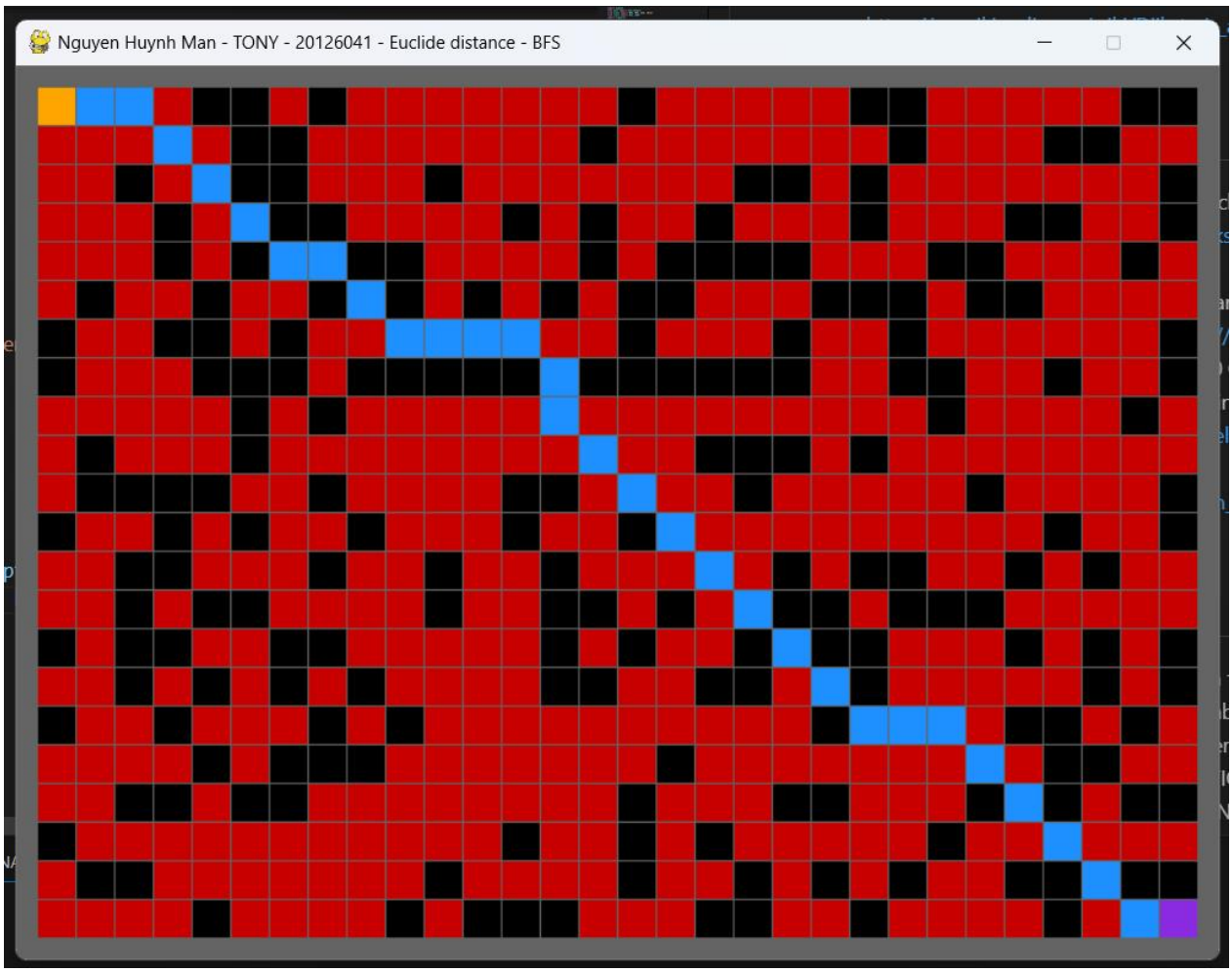
Main Idea:

- The algorithm starts from a given initial node and explores as far as possible along each branch, only backtracking when it reaches a dead-end or the goal node is found.
- It uses a stack-like approach to maintain an open set of nodes to explore and a closed set of nodes that have already been processed.
- The algorithm visually displays the exploration process by changing the color of nodes on a pygame surface (presumably used for visualization).

Explanation: - open_set is initialized with the ID of the start node. closed_set and father (an array to track parent nodes) are initialized. - The main loop continues as long as there are nodes in the open_set. - It pops the current node's ID from the open_set and retrieves the corresponding node from the graph. - If the current node is not the start or goal node, it is temporarily colored YELLOW to indicate that it is being explored. - The code checks if the current node is the goal. If yes, it prints a message and reconstructs the path from the goal to the start node. - The path is reconstructed by backtracking through the father array, and

the nodes along the path are temporarily colored BLUE. - The algorithm records the total cost, which is the number of nodes visited to reach the goal. - The execution time of the algorithm is measured using Python's time module. - If the current node is not the start or goal node, it is colored RED before proceeding. - The code then explores the neighbors of the current node. If a neighbor is not in the closed_set or the open_set, it is added to the open_set, and the father array is updated to keep track of the path. - The current node is added to the closed_set to mark it as visited.

BFS algorithm

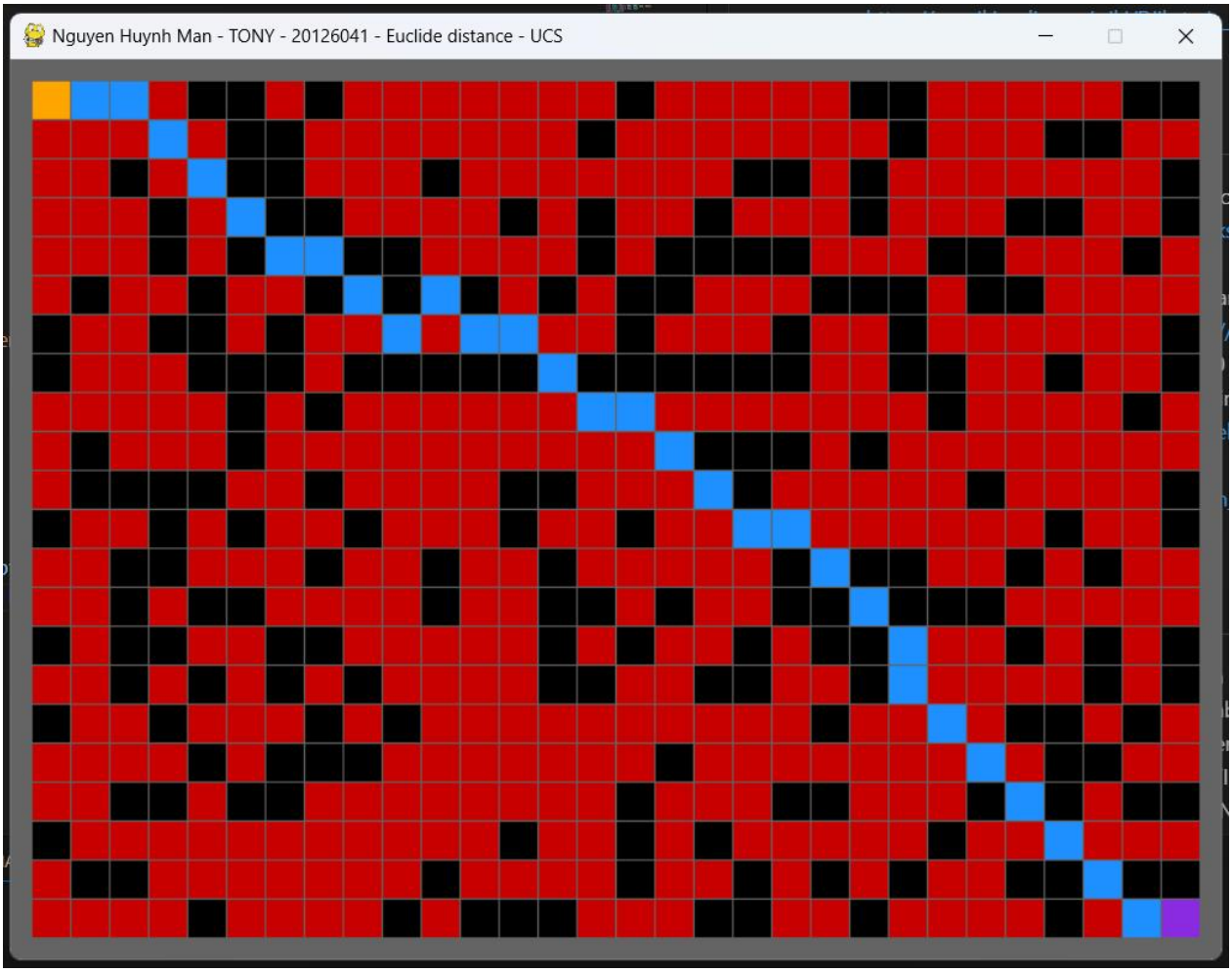


Main Idea:

- The algorithm starts from a given initial node and explores the graph level by level, similar to the way water spreads on a surface. It expands all neighbors of a node before moving on to the next level.
- The algorithm uses a queue-like approach to maintain an open set of nodes to explore and a closed set of nodes that have already been processed.
- It visually displays the exploration process by changing the color of nodes on a pygame surface (used for visualization).

Explanation: - The execution timer is started to measure the time taken by the algorithm. - The code initializes open_set with the ID of the start node, closed_set, and a father array (to track parent nodes). - The main loop continues as long as there are nodes in the open_set. - It dequeues (pops) the current node's ID from the open_set, effectively selecting the node for exploration. The selected node is retrieved from the graph. - If the current node is not the start or goal node, it is temporarily colored YELLOW to indicate that it is being explored. - The code checks if the current node is the goal. If it is, it prints a message, reconstructs the path from the goal to the start, and marks the nodes along the path as BLUE. - The reconstructed path is stored in the path list and displayed. The path is reversed to show it from the start to the goal. - The execution time of the algorithm is measured using Python's time module. - If the current node is not the start node, it is colored RED before proceeding as visited node. - The code explores the neighbors of the current node. If a neighbor is not in the open_set or the closed_set, it is enqueued (appended) to the open_set, and the father array is updated to track the path. - The current node is added to the closed_set to mark it as visited.

UCS algorithm



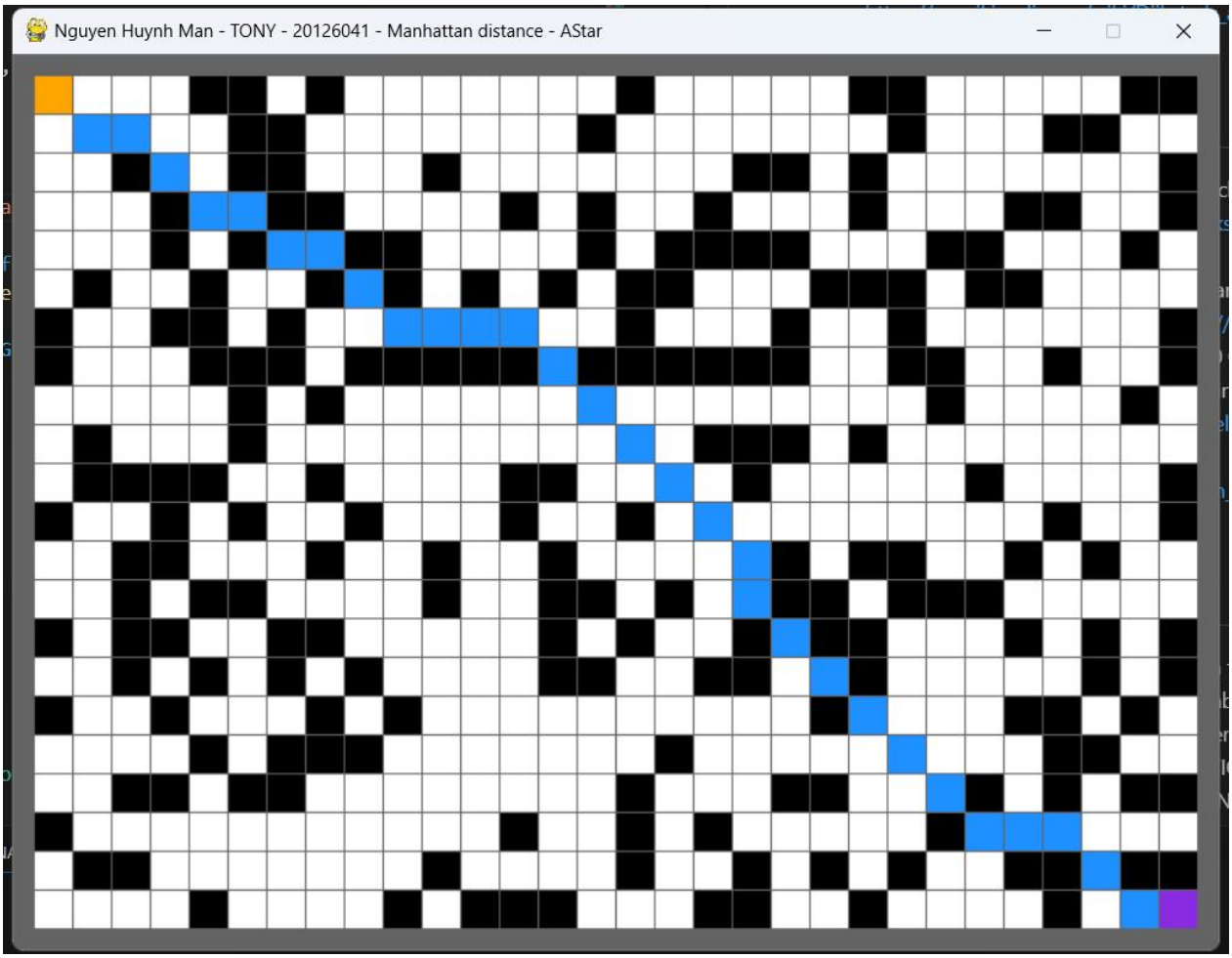
Main Idea:

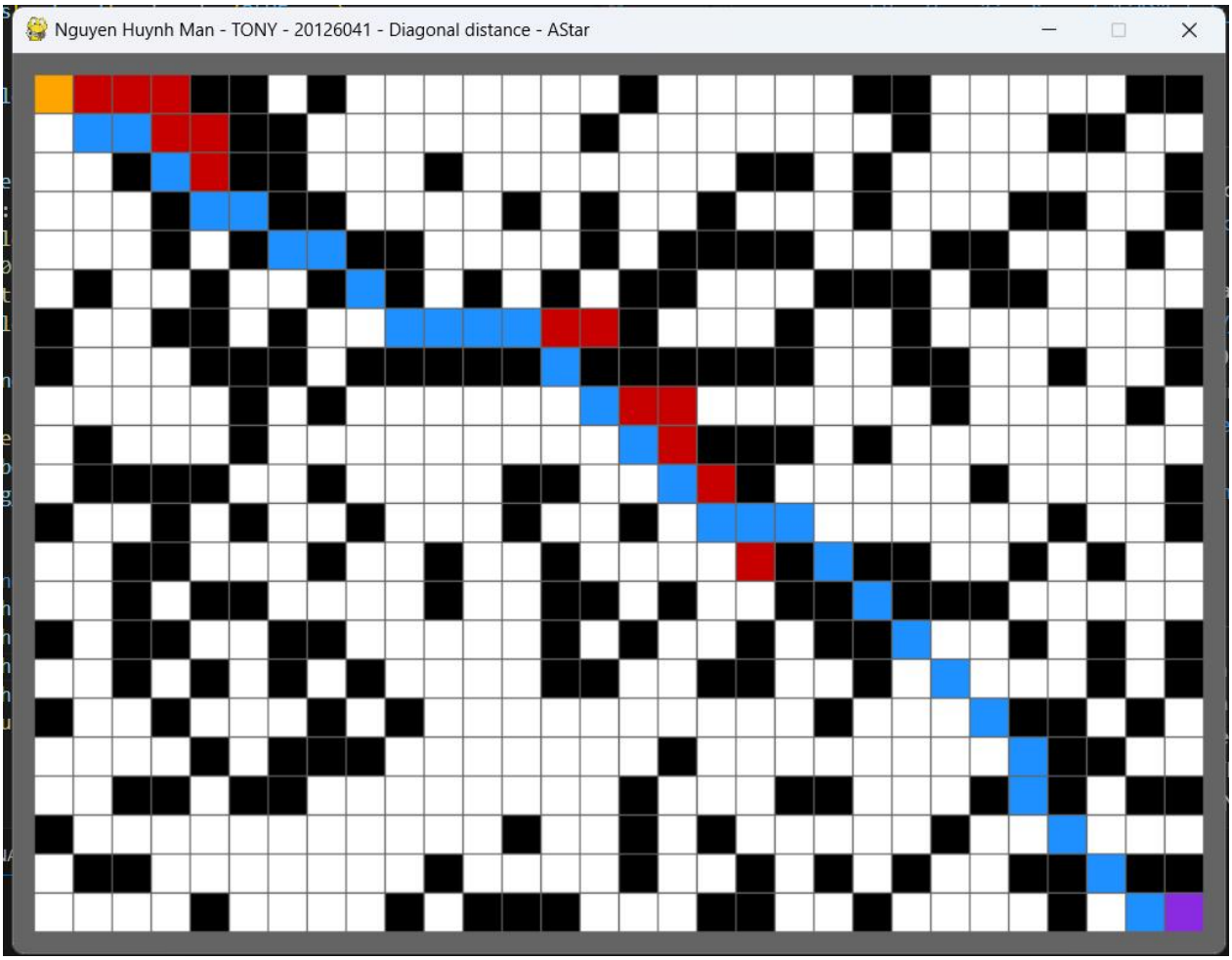
- The algorithm starts from a given initial node and explores as far as possible along each branch while maintaining the lowest path cost. It continues until it reaches the goal node.
- It utilizes a priority queue (implemented with heapq) to maintain the open set of nodes to explore and a closed set to track visited nodes.
- The code visually displays the exploration process by changing the color of nodes on a pygame surface.

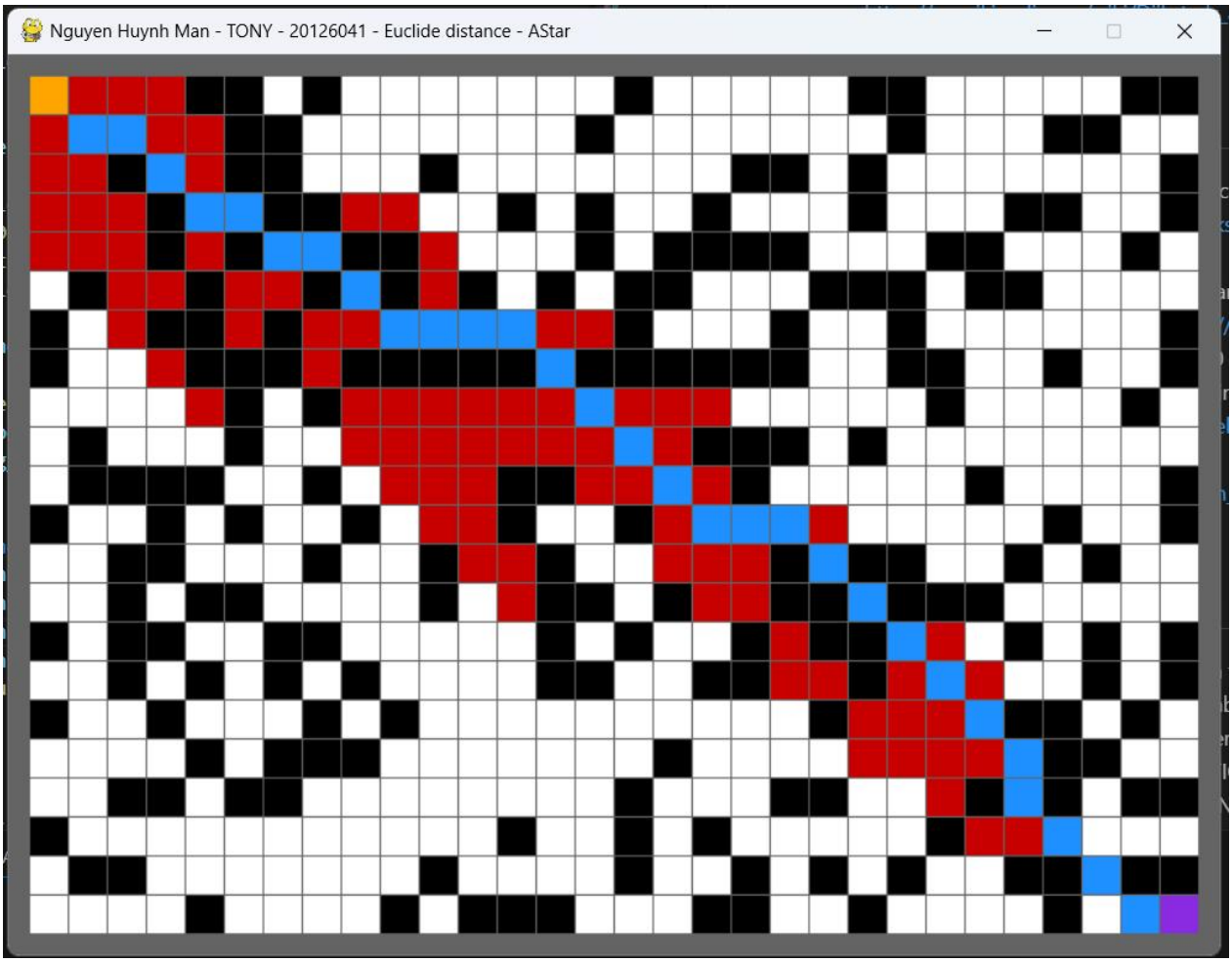
Explanation:

- The execution timer is started to measure the time taken by the algorithm.
- The code initializes open_set with a priority queue containing a tuple of the form (accumulated_cost, node_id) where accumulated_cost represents the total cost to reach the node.
- closed_set is initialized as a set to keep track of visited nodes. The father array is used to track parent nodes.
- The cost array is used to store the accumulated cost to reach each node, initialized with a large value for all nodes, except the start node with a cost of 0.
- The main loop continues as long as there are nodes in the open_set.
- The node with the lowest accumulated cost is dequeued from the priority queue using heapq.heappop.
- If the dequeued node is the goal node, the code reconstructs the path and prints the total cost.
- The path is reconstructed by backtracking through the father array, and the nodes along the path are temporarily colored BLUE.
- The execution time of the algorithm is measured using Python's time module.
- If the dequeued node is not the start node, it is temporarily colored YELLOW before proceeding.
- The code marks the dequeued node as visited by adding its ID to the closed_set.
- For each neighbor of the dequeued node, the code calculates a new cost and updates the cost and father arrays if the new cost is lower. The neighbor is enqueued to the priority queue for further exploration.
- The code proceeds to the next node in the priority queue.

A* algorithm




Nguyen Huynh Man - TONY - 20126041 - Diagonal distance - AStar



Main Idea:

- The A* algorithm starts from a given initial node, explores as far as possible along each branch while selecting nodes based on their estimated total cost to reach the goal. It backtracks when it reaches a dead-end or finds the goal node.
- It utilizes a priority queue (implemented with heapq) to maintain the open set of nodes to explore and a closed set to track visited nodes.
- The code visually displays the exploration process by changing the color of nodes on a pygame surface.

Explanation:

- The code initializes open_set with a priority queue containing a tuple of the form (estimated_total_cost, node_id) where estimated_total_cost represents the sum of the accumulated cost and the estimated remaining cost to reach the goal.
- closed_set is initialized as a set to keep track of visited nodes. The father array is used to track parent nodes.

- The g_cost array stores the accumulated cost to reach each node, initialized with float('inf') for all nodes except the start node with a cost of 0.
- The h_cost array stores the heuristic cost to reach each node, initialized with float('inf').
- The f_cost array stores the estimated total cost, initialized with float('inf').
- Three different heuristic functions are defined:
  - heuristic: Manhattan distance (L1 distance, city block distance).
  - heuristic2: Diagonal distance algorithm.
  - euclide: Euclidean distance algorithm.
- get_distance function calculates the distance between two nodes based on their coordinates. It returns 1 if they share either an x or y coordinate, and 1.4 (the square root of 2) if they do not (indicating diagonal movement).
- The main loop continues as long as there are nodes in the open_set.
- The node with the lowest estimated total cost (f-cost) is dequeued from the priority queue using heapq.heappop.
- If the dequeued node is the goal node, the code reconstructs the path and prints the total cost.
- The path is reconstructed by backtracking through the father array, and the nodes along the path are temporarily colored BLUE.
- If the dequeued node is not the start node, it is temporarily colored YELLOW before proceeding.
- The code marks the dequeued node as visited by adding its ID to the closed_set.
- For each neighbor of the dequeued node, the code calculates a new tentative g-cost, h-cost, and f-cost, and updates these values if the new cost is lower. - The neighbor is enqueued to the priority queue for further exploration.
- The code proceeds to the next node in the priority queue.

Each of the three heuristic functions provides a different way to estimate the cost to reach the goal node:

- heuristic estimates cost based on Manhattan distance (L1 distance) between the current node and the goal node.
- heuristic2 estimates cost using a diagonal distance algorithm.
- euclide estimates cost using the Euclidean distance between the current node and the goal node.

The choice of heuristic can significantly affect the performance and optimality of the A* algorithm for a specific problem.

Citation

DFS algorithm

- https://en.wikipedia.org/wiki/Depth-first_search#:~:text=Depth%2Dfirst%20search%20(DFS),along%20each%20branch%20before%20backtracking.

- favtutor.com. 2023. Depth First Search in Python (with Code) | DFS Algorithm | FavTutor. [ONLINE] Available at: https://favtutor.com/blogs/depth-first-search-python. [Accessed 20 October 2023].
- www.geeksforgeeks.org. 2012. Depth First Search or DFS for a Graph - GeeksforGeeks. [ONLINE] Available at: https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/. [Accessed 20 October 2023].

BFS algorithm

- favtutor.com. 2023. Breadth First Search in Python (with Code) | BFS Algorithm | FavTutor. [ONLINE] Available at: https://favtutor.com/blogs/breadth-first-search-python. [Accessed 20 October 2023].
- www.geeksforgeeks.org. 2012. Breadth First Search or BFS for a Graph - GeeksforGeeks. [ONLINE] Available at: https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/. [Accessed 20 October 2023].
- www.geeksforgeeks.org. 2019. Search Algorithms in AI - GeeksforGeeks. [ONLINE] Available at: https://www.geeksforgeeks.org/search-algorithms-in-ai/. [Accessed 21 October 2023].
- stackoverflow.com. 2023. What is the space complexity of Dijkstra Algorithm? - Stack Overflow. [ONLINE] Available at: https://stackoverflow.com/questions/50856391/what-is-the-space-complexity-of-dijkstra-algorithm. [Accessed 21 October 2023].

UCS algorithm

- www.youtube.com. 2023. Uniform Cost Search Algorithm | UCS Search Algorithm in Artificial Intelligence by Mahesh Huddar - YouTube. [ONLINE] Available at: https://www.youtube.com/watch?v=8ofimg8cnRE. [Accessed 20 October 2023].
- www.geeksforgeeks.org. 2019. Uniform-Cost Search (Dijkstra for large Graphs) - GeeksforGeeks. [ONLINE] Available at: https://www.geeksforgeeks.org/uniform-cost-search-dijkstra-for-large-graphs/. [Accessed 20 October 2023].
- homepage.cs.uri.edu. 2023. [ONLINE] Available at: https://homepage.cs.uri.edu/faculty/hamel/courses/2015/spring2015/csc481/lecture-notes/ln481-007.pdf. [Accessed 21 October 2023].
- www.baeldung.com. 2021. Uniform-Cost Search vs. Best-First Search | Baeldung on Computer Science. [ONLINE] Available at: https://www.baeldung.com/cs/uniform-cost-search-vs-best-first-search. [Accessed 21 October 2023].

Dijkstra

- en.wikipedia.org. 2023. Dijkstra's algorithm - Wikipedia. [ONLINE] Available at: https://en.wikipedia.org/wiki/Dijkstra's_algorithm#Practical_optimizations_and_infinite_graphs. [Accessed 20 October 2023].

Astar algorithm

- www.geeksforgeeks.org. 2016. A* Search Algorithm - GeeksforGeeks. [ONLINE] Available at: https://www.geeksforgeeks.org/a-search-algorithm/. [Accessed 20 October 2023].
- www.youtube.com. 2023. A-Star A* Search in Python [Python Maze World- pyamaze] - YouTube. [ONLINE] Available at: https://www.youtube.com/watch?v=W9zSr9jnoqY&t=595s. [Accessed 20 October 2023].
- www.baeldung.com. 2020. A* Pathfinding Algorithm | Baeldung on Computer Science. [ONLINE] Available at: https://www.baeldung.com/cs/a-star-algorithm. [Accessed 21 October 2023].
- https://en.wikipedia.org/wiki/A*_search_algorithm#:~:text=A*%20is%20an%20informed%20search,shortest%20time%2C%20etc.).

DEMO video

- www.youtube.com. 2023. [Introduction to AI - HCMUS] Lab 01 - Graph Search Algorithms - YouTube. [ONLINE] Available at: https://www.youtube.com/watch?v=6aXg1d2QXy4. [Accessed 20 October 2023].
- www.youtube.com. 2023. [INTRODUCTION TO ARTIFICIAL INTELLIGENCE - HCMUS] LAB 01 – SEARCH IN GRAPH - YouTube. [ONLINE] Available at: https://www.youtube.com/watch?v=8m7uWyaze5A. [Accessed 20 October 2023].