VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION AND TECHNOLOGY

23CLC08

REPORT

PROJECT 1 - SEARCH
Let's chase Pac-Man!

INSTRUCTOR

Nguyen Thanh Tinh

—o0o—

STUDENT

23127318 - Le Ho Dan Anh
23127284 - Nguyen Ngoc Mai Xuan
23127488 - Le Thi Minh Thu
23127361 - Thach Ngoc Han

HO CHI MINH CITY, March 2025

Content

1 Project Planning and Task Distribution **2**
1.1 Completion Levels . . . 2
1.2 Task Distribution . . . 2

2 Algorithm Description: **2**
2.1 Breadth-First Search (BFS) . . . 2
2.2 Depth-First Search (DFS) . . . 4
2.3 Uniform-Cost Search (UCS) . . . 4
2.4 A* Search (A*) . . . 7

3 Experiments: **11**

4 Analyze and evaluate algorithms **31**
4.1 Breadth-First Search (BFS) . . . 32
4.1.1 Analysis of Results . . . 32
4.1.2 Explanation . . . 33
4.2 Depth-First Search (DFS) . . . 33
4.2.1 Analysis of Results . . . 33
4.2.2 Explanation . . . 33
4.3 Uniform Cost Search (UCS) . . . 33
4.3.1 Analysis of Results . . . 33
4.3.2 Explanation . . . 33
4.4 A* Algorithm . . . 34
4.4.1 Analysis of Results . . . 34
4.4.2 Explanation . . . 34
4.5 Summary Table . . . 34
4.6 General Conclusion . . . 34

5 Video Game **35**

1 Project Planning and Task Distribution

1.1 Completion Levels

Level	Completion
1	100%
2	100%
3	100%
4	100%
5	100%
6	100%

1.2 Task Distribution

Task Category	Task	Assigned Person
Algorithm Implementation	Implement level 1, 2	Nguyen Ngoc Mai Xuan
	Implement level 3, 4	Le Thi Minh Thu
	Implement level 5, 6	Le Ho Dan Anh
User Interface	Menu, start screen, game over screen	Thach Ngoc Han
	Main game screen, map selection	Le Ho Dan Anh
Map Design	Map for level 1, 2, 3, 4, 5	Nguyen Ngoc Mai Xuan
	Map for level 6	Thach Ngoc Han
Report	Compilation, presentation	Le Thi Minh Thu

Table 1: Task Allocation

Assigned Person	Student ID	Total Completion
Nguyen Ngoc Mai Xuan	23127284	100%
Le Ho Dan Anh	23127318	100%
Thach Ngoc Han	23127361	90%
Le Thi Minh Thu	23127488	90%

Table 2: Total Completion per Person

2 Algorithm Description:

2.1 Breadth-First Search (BFS)

- Determine initial position and target: Convert the coordinates of the ghost and Pac-Man from pixels to indices (GRID_SIZE).
- Check if the previously calculated path is valid: If the ghost has already found a path before and Pac-Man hasn't moved yet, reuse the old path to save calculation time.
- BFS loop:

  - At each iteration, BFS takes the current path (path) from the head of the queue.
  - Take the last cell in the path ((x, y)) to expand further.
  - Save the number of expanded_nodes.

- Check if go to Pac-man:
  - If the current cell matches Pac-Man, the ghost has found the shortest path.
  - Save the result (self.path = path) and note the time taken, number of nodes expanded, and memory used.
  - Stop the algorithm and return the path.

- Create child nodes (4 cells the ghost can go to):
  - Iterate through 4 directions: right, down, left, up.
  - Calculate the coordinates of the new cell the ghost can move to.

- Check if the new cell is valid:
  - The new cell must be in the map `(0 <= next_x < len(self.map) and 0 <= next_y < len(self.map[0]))`.
  - The new cell is not a wall or an obstacle.
  - The new cell is not in visited.

- If the new cell is Pac-Man, return the path immediately to save resources.

- If Pac-Man is not found:
  - Add new path to queue.
  - Mark new cell as visited.

- If BFS runs out without finding Pac-Man: return empty path.

- Pseudocode:

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node, or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier) /* chooses the shallowest node in frontier */
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure
```

Figure 1: BFS

2.2 Depth-First Search (DFS)

- Determine Initial Position and Target
  - The ghost's and Pac-Man's positions are converted from pixel coordinates to grid indices using GRID_SIZE.
  - The start position is the ghost's current location.
  - The end position is Pac-Man's location.
- DFS Loop:
  - The loop runs until the stack is empty.
  - The last node (deepest path) is popped from the stack for exploration.
- Check if Pac-Man is Found: If Pac-Man is found, DFS terminates early and returns the path.
- Expand Neighboring Nodes (Ghost's Possible Moves):
  - The ghost moves in four directions: right, down, left, up.
  - It only moves to valid tiles:
    - The tile is within the grid boundaries.
    - It is not a wall (1) or obstacle (4).
    - It has not been visited before.
- Early Stopping Optimization: If Pac-Man is found during expansion, the search stops immediately.
- If No Path is Found: If the stack is empty and Pac-Man was not reached, return an empty path.

2.3 Uniform-Cost Search (UCS)

- **Idea:**
  - UCS expands the node with the smallest cost $g\_n(x, y)$ from the root, similar to Dijkstra's algorithm. In this algorithm, the *g_cost* function calculate the cost from the current node to its successor nodes is crucial.
- **Defining the $g\_cost(x, y)$ function**
  - The function $g\_cost(x, y)$ calculates the cost from the current node to the successor node. Where (x, y) is the coordinate of the Ghost's position.
  - **Base cost:** Moving from one node to another has a cost of 1.
  - **Special case:** When the Ghost moves near a wall, its movement directions become limited. This restriction increases the risk of getting trapped if Pac-Man changes direction. By increasing the cost to 2, the algorithm discourages the Ghost from moving near walls unless no other options are available.

$$g\_cost(x, y) = 2$$

- **Normal case:**
  - Define the Manhattan distance from the Ghost's position to Pacman's position $(\text{end}[0], \text{end}[1])$:

$$\text{distance\_to\_pacman} = |x - \text{end}[0]| + |y - \text{end}[1]|$$

  - Define **unit** as the maximum Manhattan distance between two points on the board:

$$\text{unit} = \frac{\text{WIDTH}}{\text{GRID\_SIZE}} + \frac{\text{HEIGHT}}{\text{GRID\_SIZE}} - 2$$

  - If Pacman has a **power-up** (meaning he can eat the Ghost), the Ghost must move as far away from Pacman as possible to avoid being eaten. In this case, the cost function is defined as:

$$g\_n(x, y) = \text{base\_cost} + \text{avoid\_cost}$$

    where the avoidance cost is defined as:

$$\text{avoid\_cost} = \left(1 - \frac{\text{distance\_to\_pacman}}{\text{unit}}\right)$$

    Since *unit* is the maximum distance between two points on the board, the value $\frac{\text{distance\_to\_pacman}}{\text{unit}}$ is always within $[0, 1]$. The larger the distance to Pacman, the smaller the avoid_cost, which makes sense because the Ghost has a higher chance of avoiding being eaten.
  - In the best case, $g\_n(x, y) = \text{base\_cost}$ (when $\text{avoid\_cost} = 0$).
  - If Pacman does **not** have a power-up, the Ghost tries to approach Pacman. In this case:

$$g\_cost(x, y) = \text{base\_cost} + \frac{\text{distance\_to\_pacman}}{\text{unit}}$$

    If distance_to_pacman is small, $g\_cost(x, y)$ is small, and vice versa. This ensures that the Ghost expands nodes that have the highest chance of reaching Pacman as quickly as possible with the least cost.

- **Implementation:**
  - **Step 1: Initialize Initial Values**
    - Determine the starting position of the Ghost (start) and the target position (end) based on PacMan's coordinates.
    - Initialize the priority queue `pq` containing the tuple (cost, position, path) with the starting values.
    - Create the `visited` set to store visited positions.
    - Maintain an `expanded` list to track expanded nodes.
    - Define `max_pq_size` to record the largest size of the priority queue during execution.
    - Define `expanded_nodes` to count the number of expanded nodes.
  - **Step 2: Define Cost Function** $g\_cost(x, y)$
    - Base cost: Each movement step has a base cost of 1.
    - If PacMan has a power-up, Ghost avoids PacMan by adding an avoidance cost `avoid_pacman`.

* If PacMan has no power-up, Ghost attempts to approach the target using the normalized distance `distance_to_pacman/unit`.
* If Ghost is near a wall and not close to PacMan, the movement cost increases to 2 to discourage moving close to walls.

– **Step 3: Path Search Using UCS**

* Traverse the priority queue `pq` in order of the lowest cost.
* If position (x, y) has been expanded, skip it.
* If position (x, y) is the goal (end), store path information, execution time, memory usage, and the number of expanded nodes, then terminate.
* If not at the goal, expand neighboring cells $(next_x, next_y)$ in four directions (left, right, up, down) if valid (not a wall or obstacle).

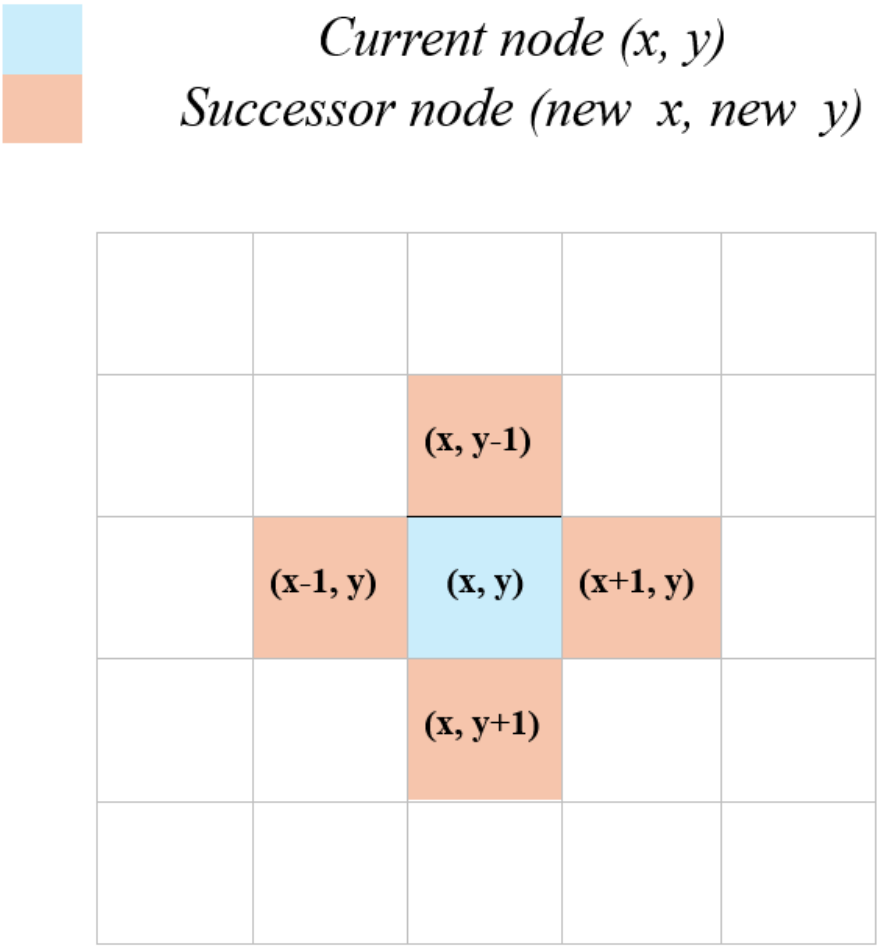

Figure 2: 4 Directions

* Compute `new_cost` for the new movement by adding `g_cost(x, y)`, then push into the priority queue `pq`.

– **Step 4: Algorithm Termination**

* If a path is found, display execution time, memory usage, the number of expanded nodes, and store the path in `self.path`.
* If no path is found, display a failure message and store information on expanded nodes and memory usage.

- **Step 5: Performance Evaluation**
  - Execution time is measured as the difference between `start_time` and the final execution time.
  - Memory usage is calculated as the total size of the `visited` set and the maximum `max_pq_size` recorded.
  - The final results include the computed path, execution time, memory usage, and the number of expanded nodes.

- **Pseudocode:**

```
FUNCTION move_ucs():
    start_time ← current_time()

    start, end ← initial and target positions on the grid
    pq ← priority_queue storing (cost, position, path)
    visited ← set of visited cells
    expanded_nodes, max_pq_size ← 0, 0

    FUNCTION g_cost(x, y):
        Compute movement cost based on distance to Pacman and walls
        RETURN cost

    WHILE pq is not empty:
        Extract the lowest-cost cell from pq
        IF already visited → skip
        Mark as visited, update expanded nodes count

        IF reached the goal → save path, return result

        FOR each direction (left, right, up, down):
            IF valid, unvisited cell → compute cost, push into pq

    IF no path found → print failure message
    RETURN []
```

Figure 3: UCS

2.4 A* Search (A*)

- **Idea:** The A* algorithm finds the optimal path using the evaluation function:

$$f_n(x, y) = g_n(x, y) + h_n(x, y)$$

where:

- $g_n(x, y)$ is the cost to move from the start position to the Ghost's position (x, y).
- $h_n(x, y)$ is the heuristic function estimating the cost from the Ghost's position to Pacman.

- **Defining the $g_n(x, y)$ and the heuristic $h_n(x, y)$**
  - **The $g_n(x, y)$ function**
    - The function $g_n(x, y)$ depends on the cost $g\_cost(x, y)$ between two adjacent nodes (from parent node to child node), which is defined in the same way as in UCS.
  - **The heuristic $h_n(x, y)$ function**
    - The heuristic function $h_n(x, y)$ is the Manhattan distance from the Ghost's position (x, y) to Pacman's position $(\text{end}[0], \text{end}[1])$:

$$h_n(x, y) = |x - \text{end}[0]| + |y - \text{end}[1]|$$

$$\text{Manhattan Distance} = |x - a| + |y - b|$$

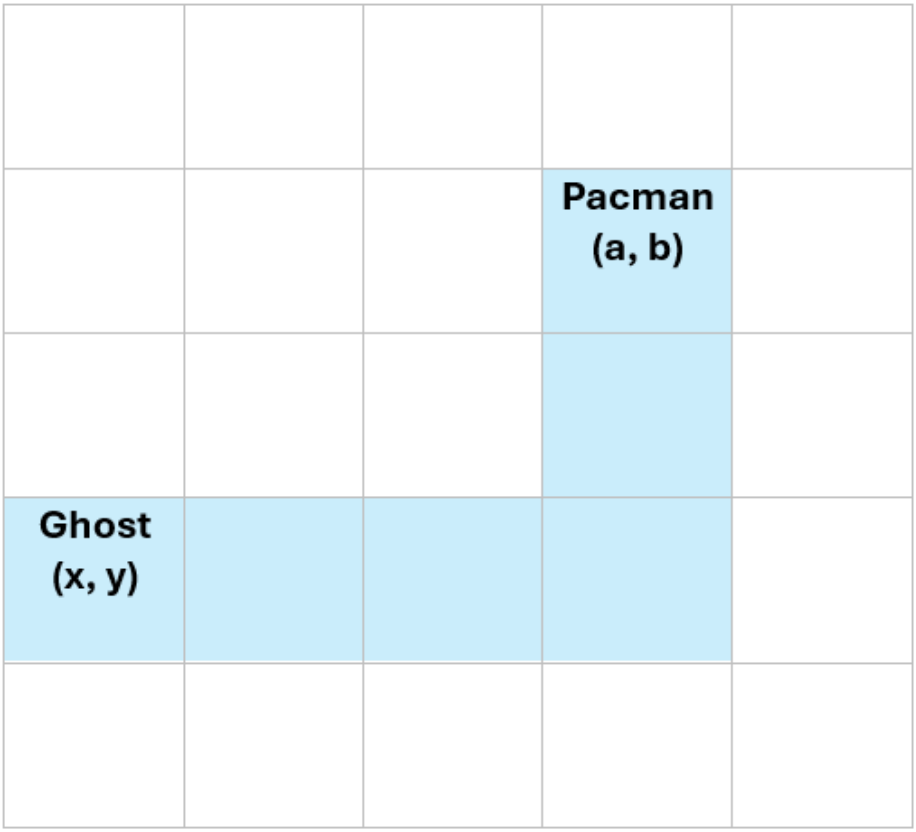

Figure 4: Manhatta Distance

- **Implementation:**
  - **Step 1: Initialization**
    - **Determine Start and Goal Positions:** Identify the starting position of the Ghost (`start`) and the target position (`end`) based on Pac-Man's coordinates.
    - **Initialize the Priority Queue (`pq`):** Store tuples in the form (cost, position, path), where:

· cost:$f(n) = g(n) + h(n)$, combining the actual cost and heuristic estimate.

· position represents the current node's coordinates.

· path maintains the sequence of moves taken.

* **Create a `visited` Set:** This set keeps track of visited positions to avoid redundant exploration.

* **Maintain an `expanded` List:** Tracks all expanded nodes during execution for analysis purposes.

* **Define `max_pq_size`:** Records the largest size of the priority queue at any point during execution to measure memory usage.

* **Define `expanded_nodes` Counter:** Counts the number of nodes expanded during the search process, providing insight into algorithm efficiency.

– **Step 2: Defining the heuristic $h_n(x, y)$ and the $g_cost(x, y)$ function**

* Having been discussed above.

– **Step 3: Path search using A***

* While the priority queue is not empty, repeat the following steps:

· Extract the node with the lowest f-value from the priority queue.

· If the node has already been visited, skip it.

· Otherwise, mark the node as visited and expand it.

· If the extracted node is the goal, return the computed path.

· Explore all four possible moves (left, right, up, down):

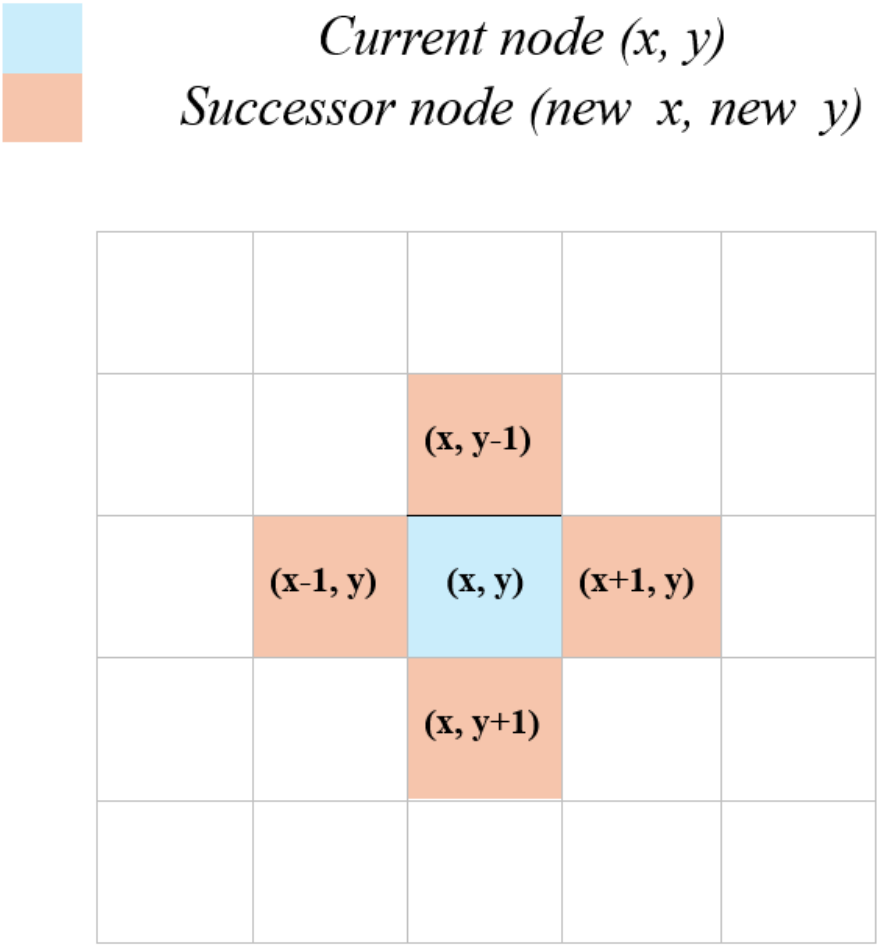

Figure 5: 4 Directions

- For each move, check if it is valid (i.e., not a wall or obstacle) and has not been visited.
- If valid, compute the new cost values:
- $new\_g = g + g\_cost$ (cost from the start node to the current node)
- $new\_f = new\_g + h$ (total estimated cost, where h is the heuristic estimate to the goal)
- If this new path improves the existing cost to this node, update the cost and push the node into the priority queue.

– **Step 4: Termination**

* If the queue is empty, return no path.
* Print execution time, memory usage, and expanded nodes.

– **Step 5 : Performance Evaluation**

* Execution time is measured as the difference between `start_time` and the final execution time.
* Memory usage is calculated as the total size of the `visited` set and the maximum `max_pq_size` recorded.

* The final results include the computed path, execution time, memory usage, and the number of expanded nodes.

- **Pseudocode:**

```
FUNCTION move_astar():
    start_time ← current_time()

    start, end ← initial and target positions on the grid
    pq ← priority_queue storing (f_cost, position, path)
    g_n, f_n ← cost dictionaries
    visited ← set of visited cells
    expanded_nodes, max_pq_size ← 0, 0

    FUNCTION heuristic(x, y):
        RETURN Manhattan distance to goal

    FUNCTION g_cost(x, y):
        Compute movement cost based on distance to Pacman and walls
        RETURN cost

    WHILE pq is not empty:
        Extract the lowest-cost node from pq
        IF already visited → skip
        Mark as visited, update expanded nodes count

        IF reached the goal → save path, return result

        FOR each direction (left, right, up, down):
            IF valid, unvisited cell → compute g_cost, update f_cost, push into pq

    IF no path found → print failure message
    return result
```

Figure 6: A*

3 Experiments:

- **Test case 1**
  - **Ghost Coordinate** (1,1)
  - **Pacman Coordinate** (21,21)
  - **BFS**

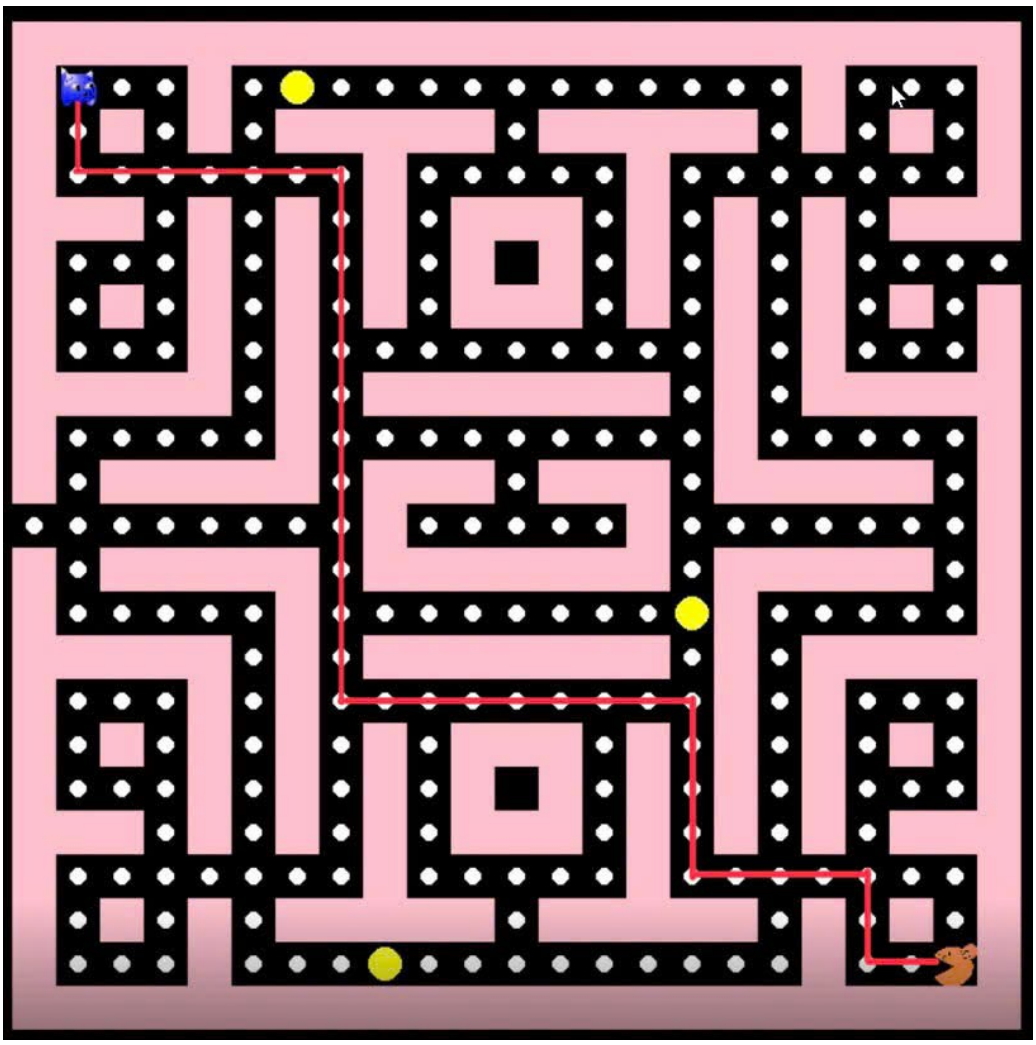

Figure 7: BFS

* **Search Time:** 0.001940966s

* **Memory Used:** 9168 bytes

* **Expanded Nodes:** 251

– **DFS**

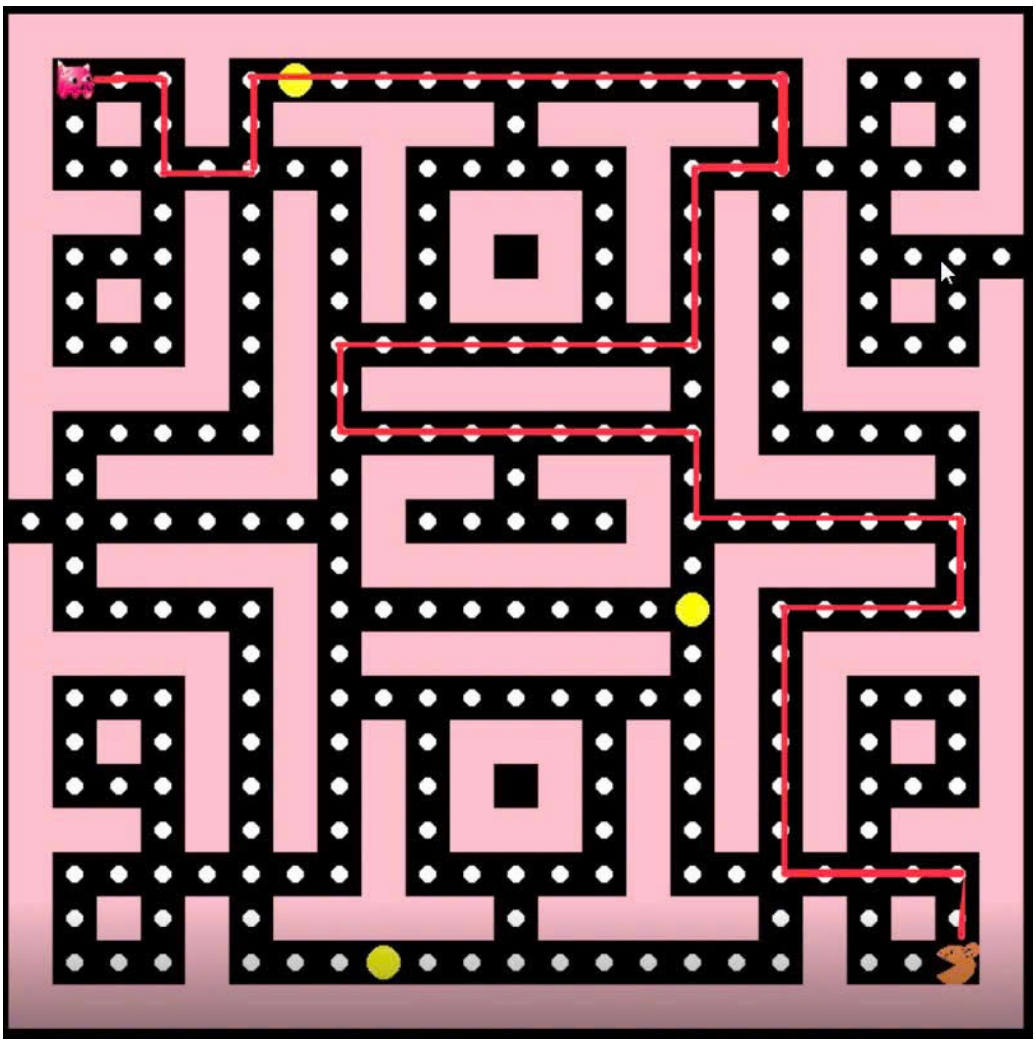

Figure 8: DFS

* **Search Time:** 0.000963211s

* **Memory Used:** 8720 bytes

* **Expanded Nodes:** 217

– **UCS**

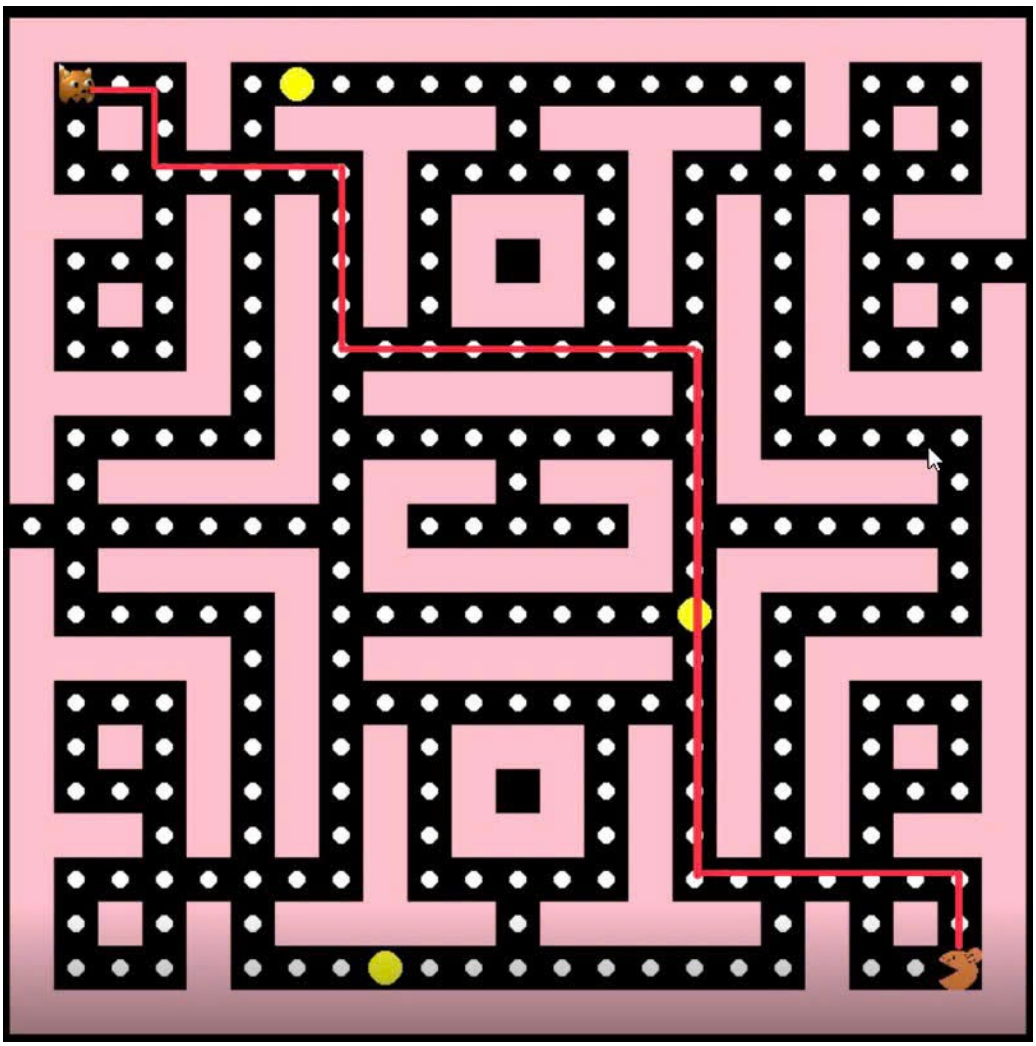

Figure 9: UCS

* **Search Time:** 0.000520706s

* **Memory Used:** 8624 bytes

* **Expanded Nodes:** 255

– **A***

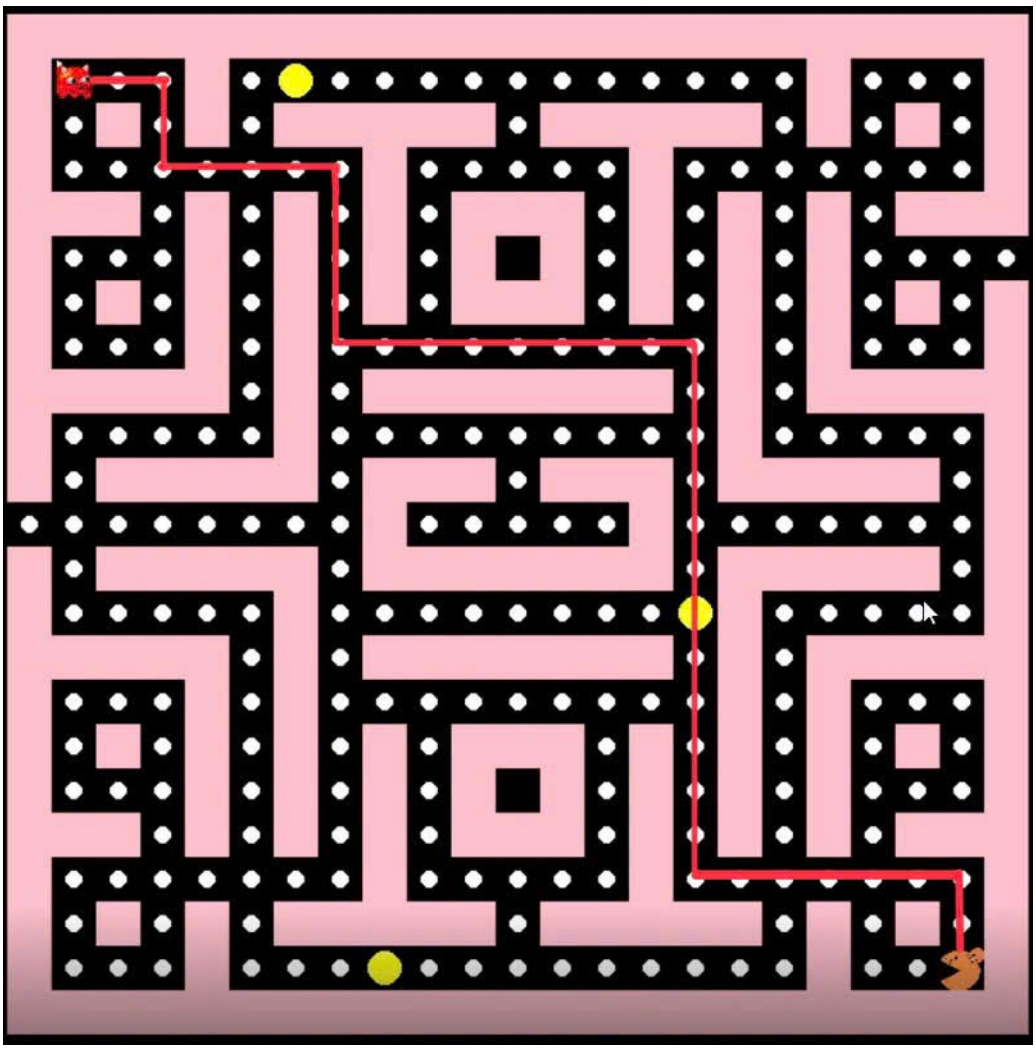

Figure 10: A 1

* **Search Time:** 0.001005650
* **Memory Used:** 8424 bytes
* **Expanded Nodes:** 141

- **Test case 2**
  - **Ghost Coordinate** (10, 1)
  - **Pacman Coordinate** (1, 15)
  - **BFS**

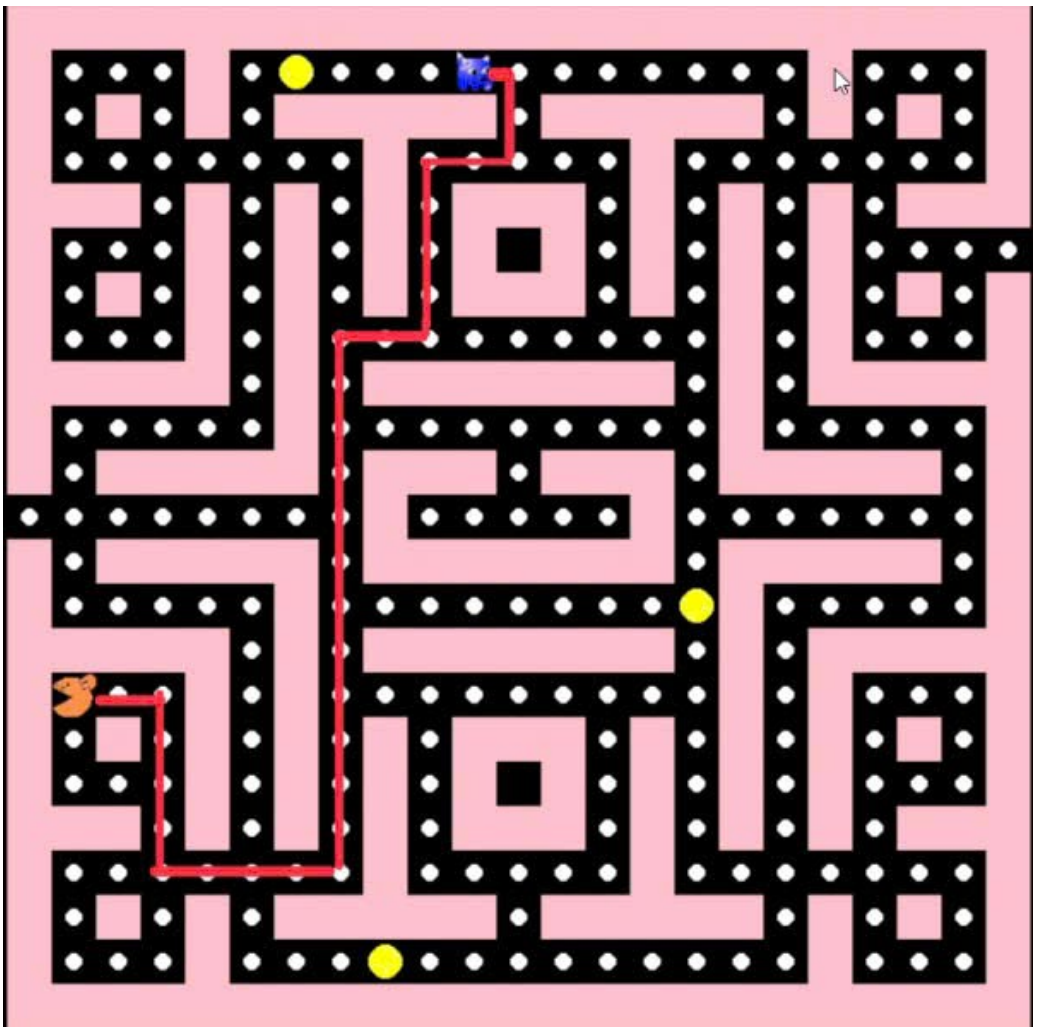

Figure 11: BFS

* **Search Time:** 0.00143027305s

* **Memory Used:** 9168 bytes

* **Expanded Nodes:** 257

– **DFS**

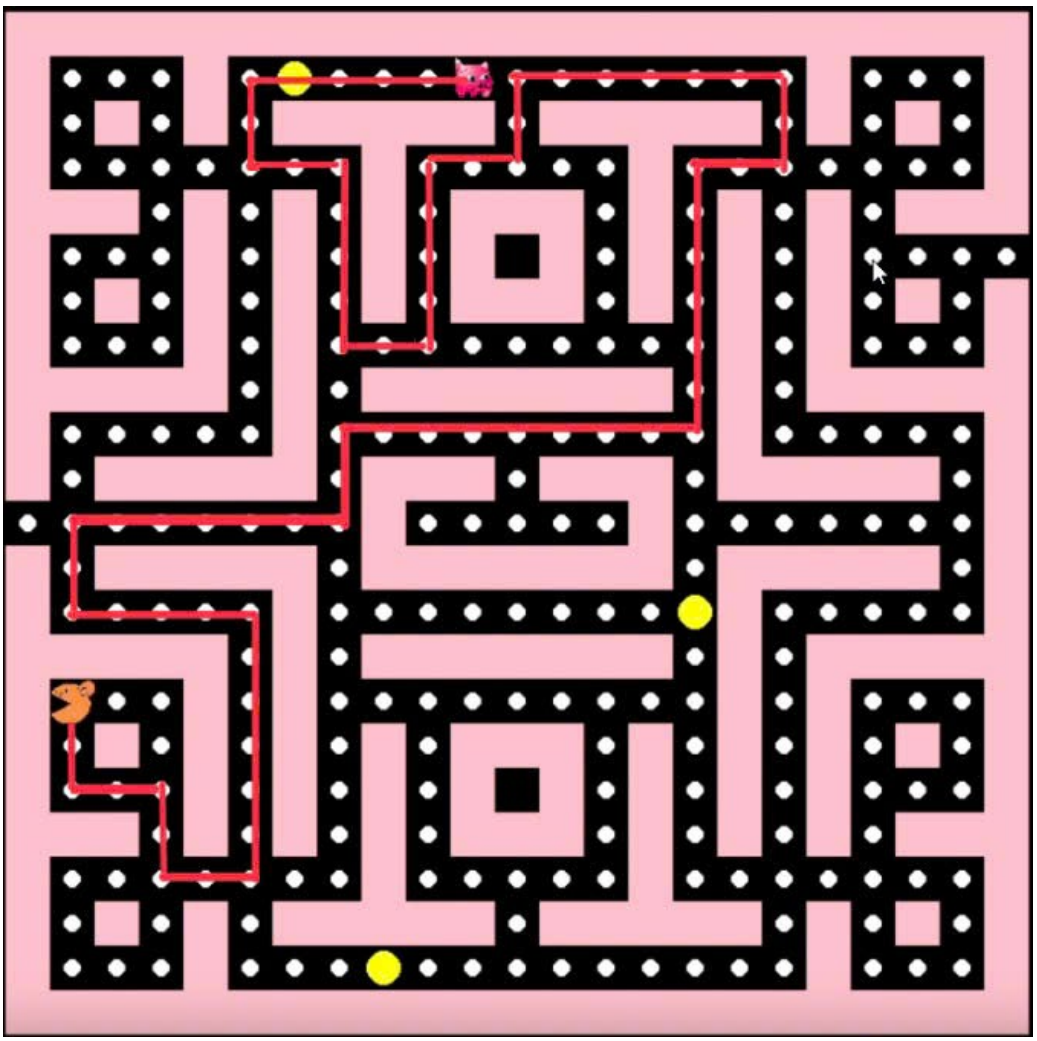

Figure 12: DFS

* **Search Time:** 0.000892878 s

* **Memory Used:** 8624 bytes

* **Expanded Nodes:** 125

– **UCS**

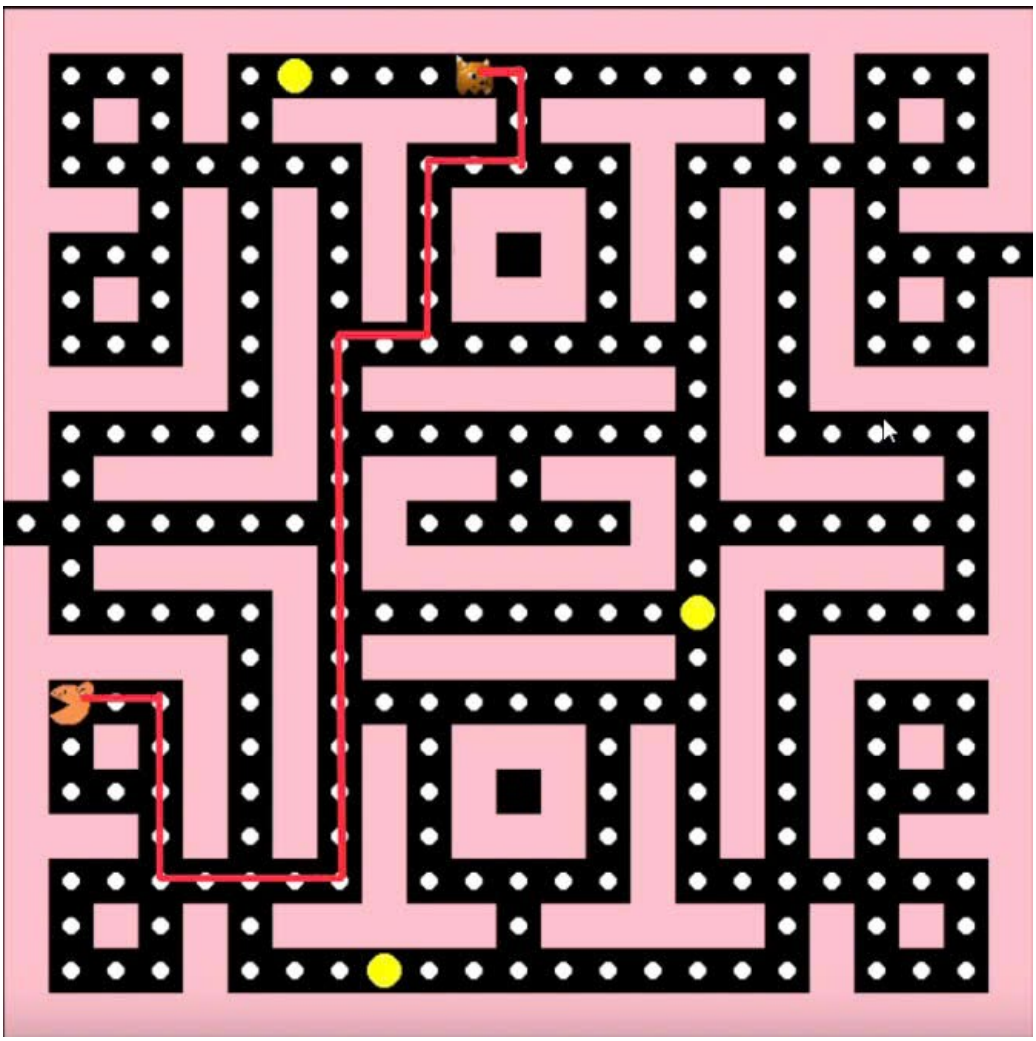

Figure 13: UCS

* **Search Time:**0.000960588s

* **Memory Used:** 8592 bytes

* **Expanded Nodes:** 249

– **A***

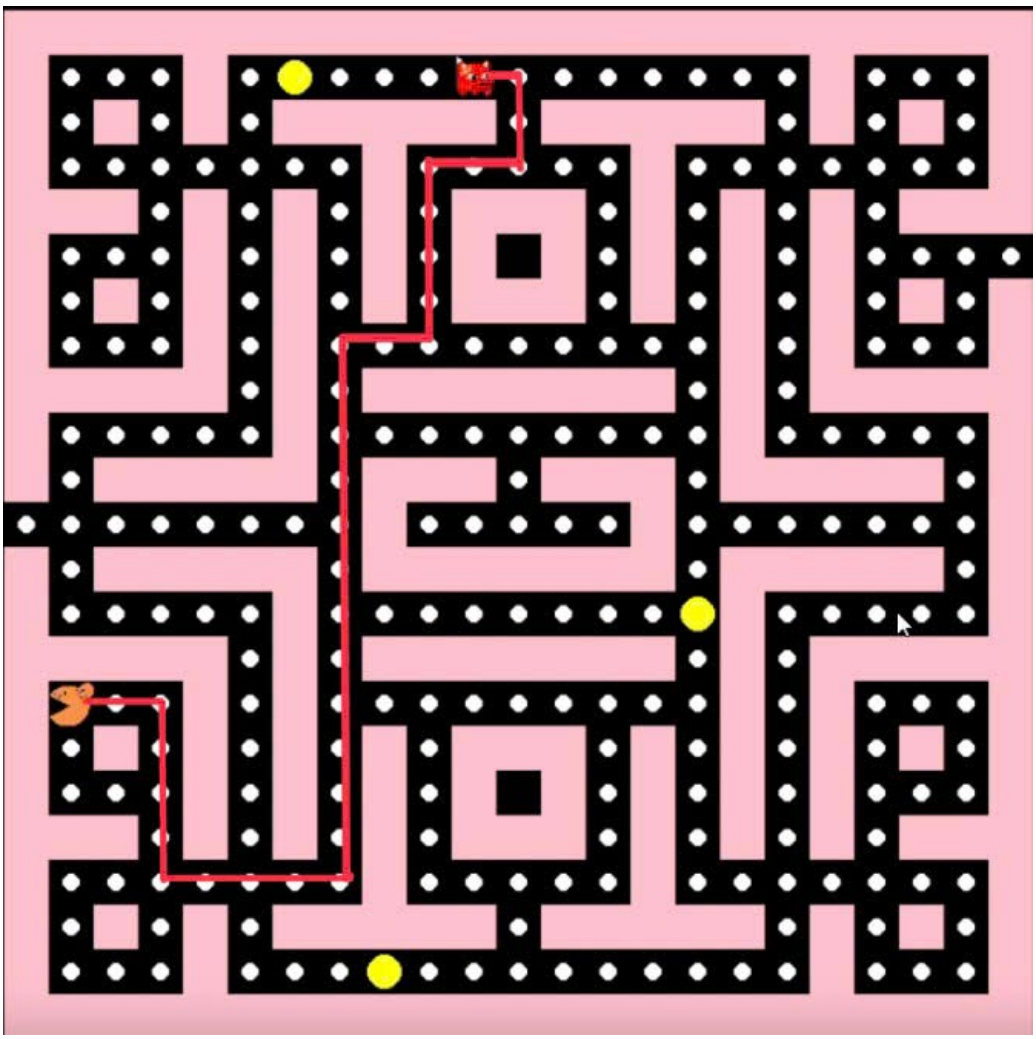

Figure 14: A 1

  * **Search Time:** 0.000986099s
  * **Memory Used:** 8424 bytes
  * **Expanded Nodes:** 140

- **Test case 3**
  - **Ghost Coordinate** (22, 5)
  - **Pacman Coordinate** (9, 11)
  - **BFS**

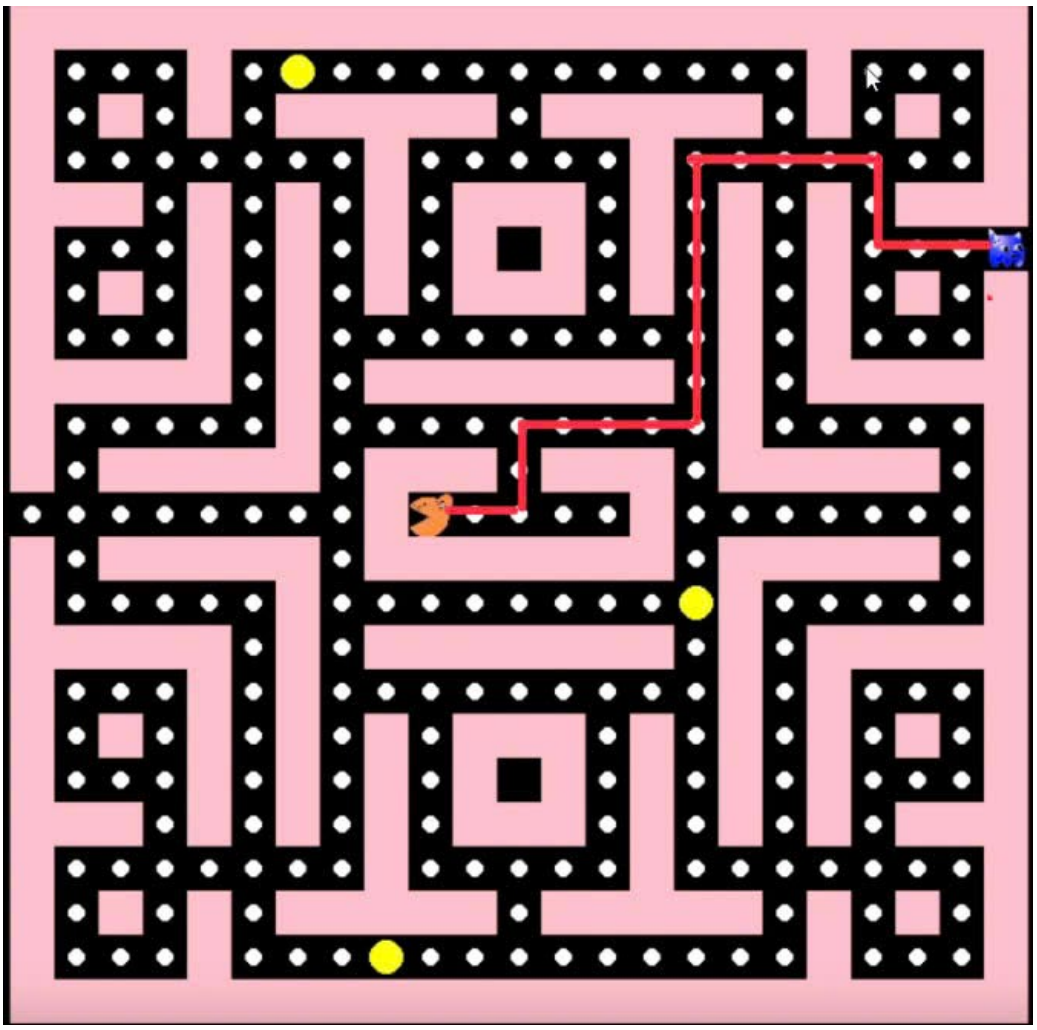

Figure 15: BFS

* **Search Time:** 0.001305103s

* **Memory Used:** 9168 bytes

* **Expanded Nodes:** 104

– **DFS**

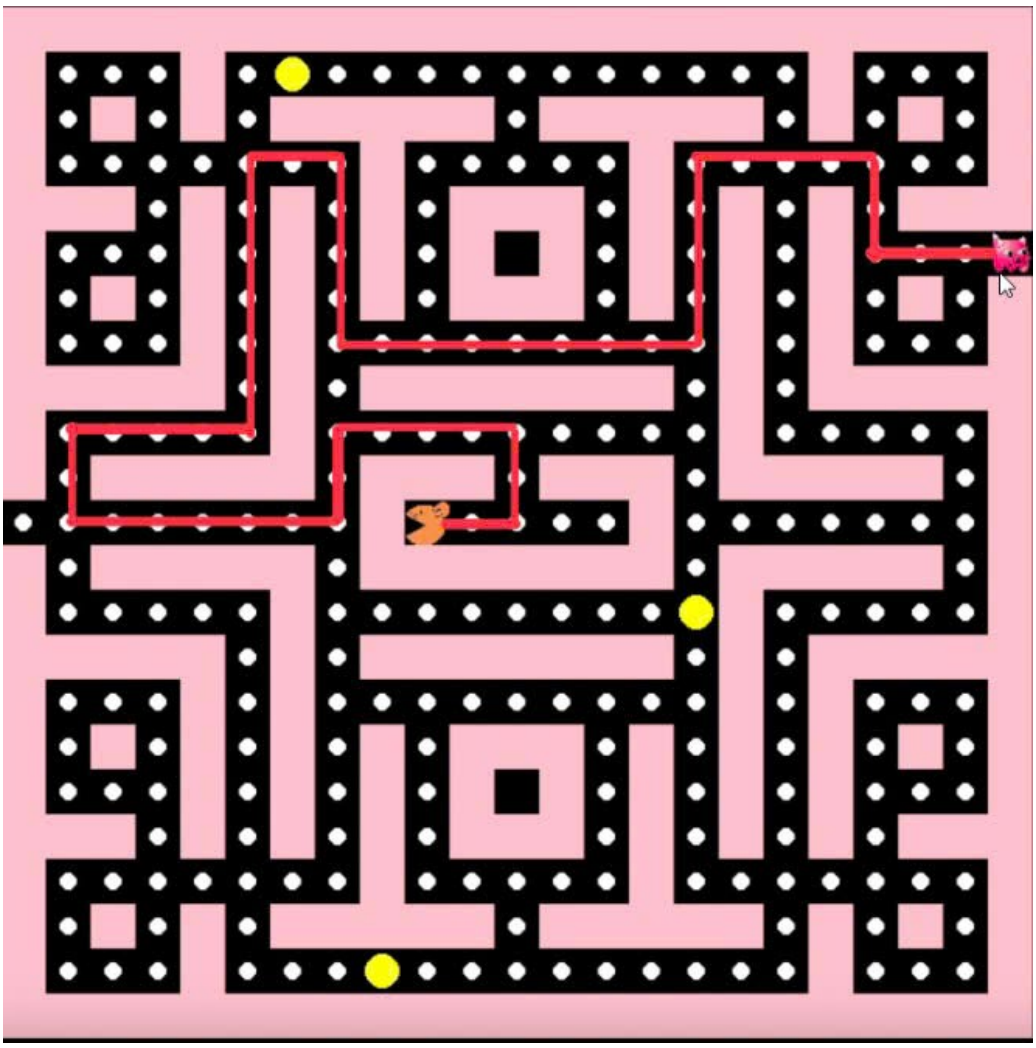

Figure 16: DFS

* **Search Time:** 0.011940956s

* **Memory Used:** 8656 bytes

* **Expanded Nodes:** 256

– **UCS**

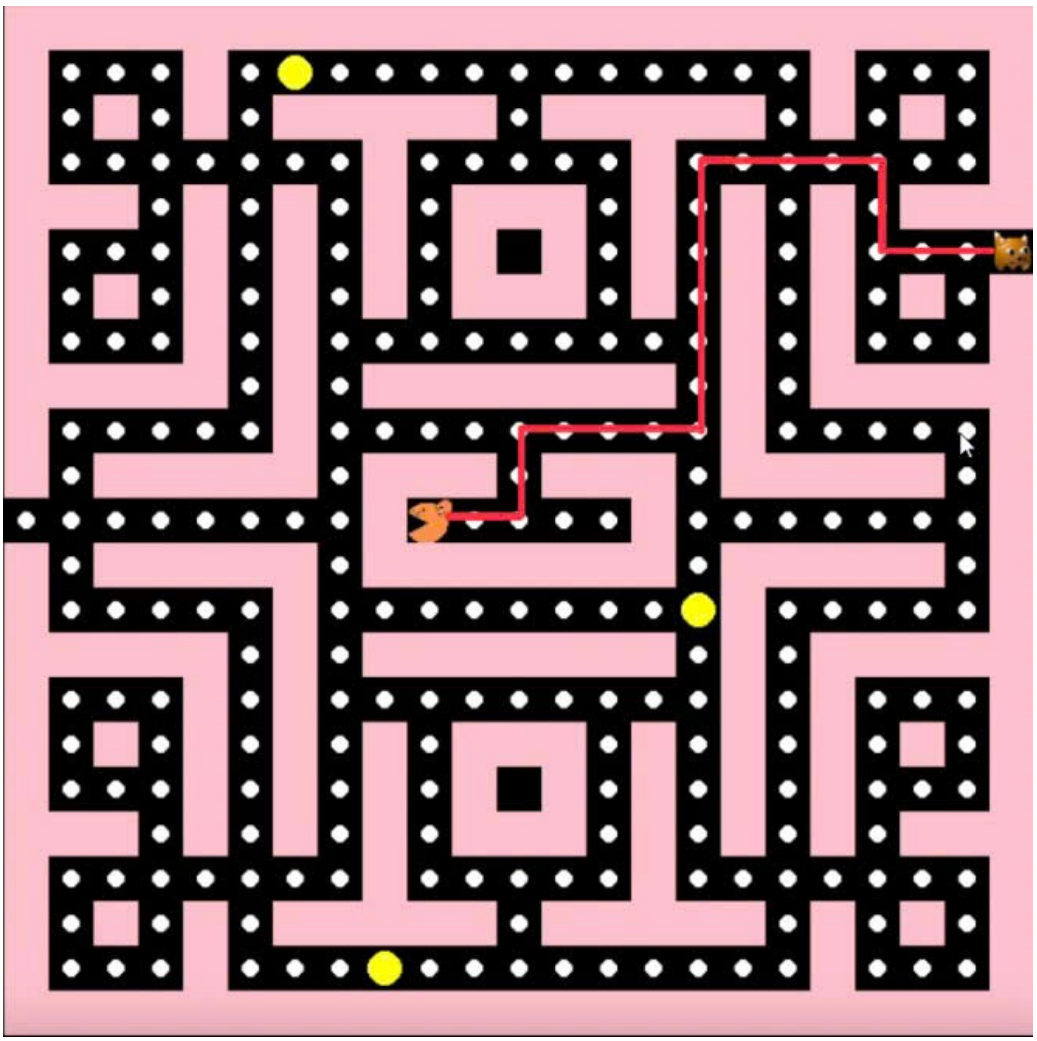

Figure 17: UCS

* **Search Time:** 0.000392437s

* **Memory Used:** 8592 bytes

* **Expanded Nodes:** 107

– **A***

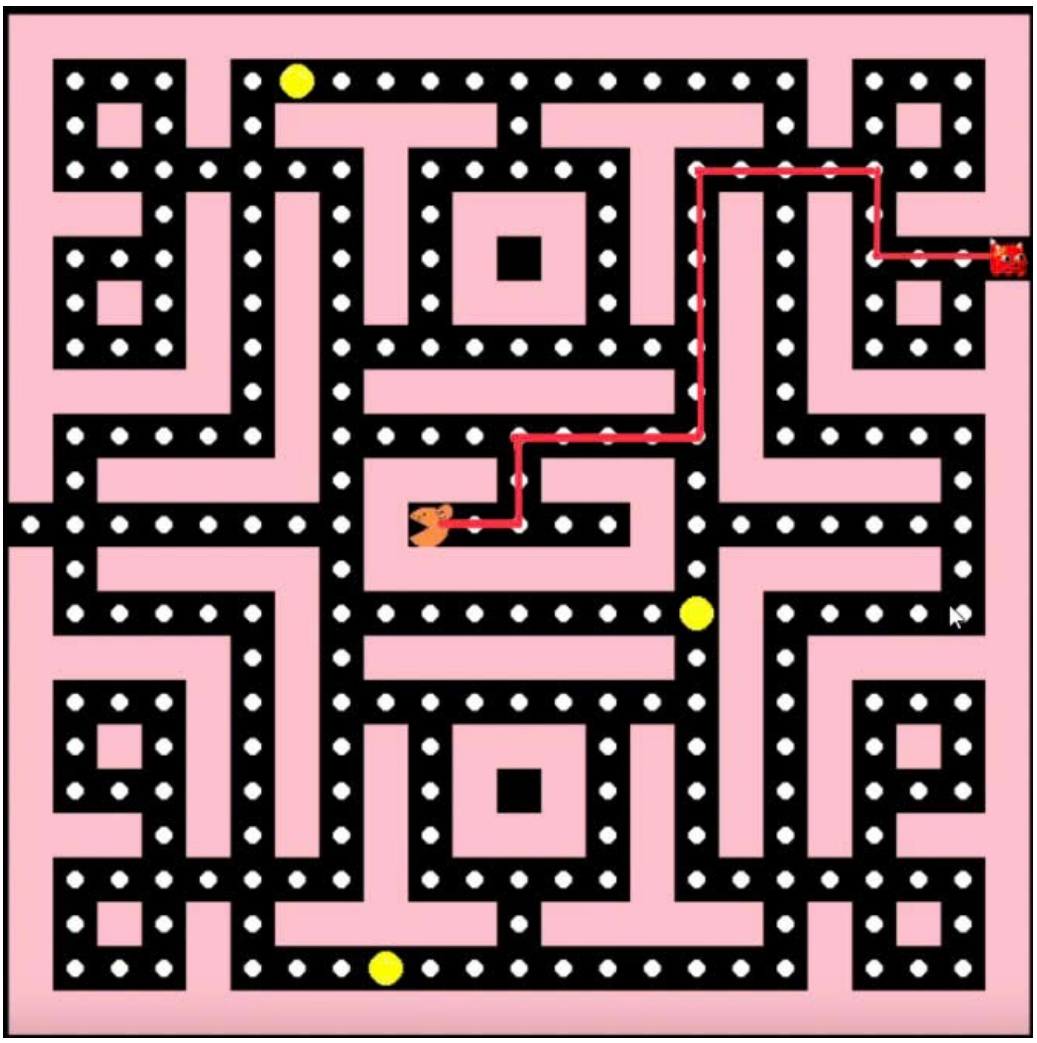

Figure 18: A*

  * **Search Time:** 0.000247002s
  * **Memory Used:** 2276 bytes
  * **Expanded Nodes:** 45

- **Test case 4**
  - **Ghost Coordinate** (11, 21)
  - **Pacman Coordinate** (11, 11)
  - **BFS**

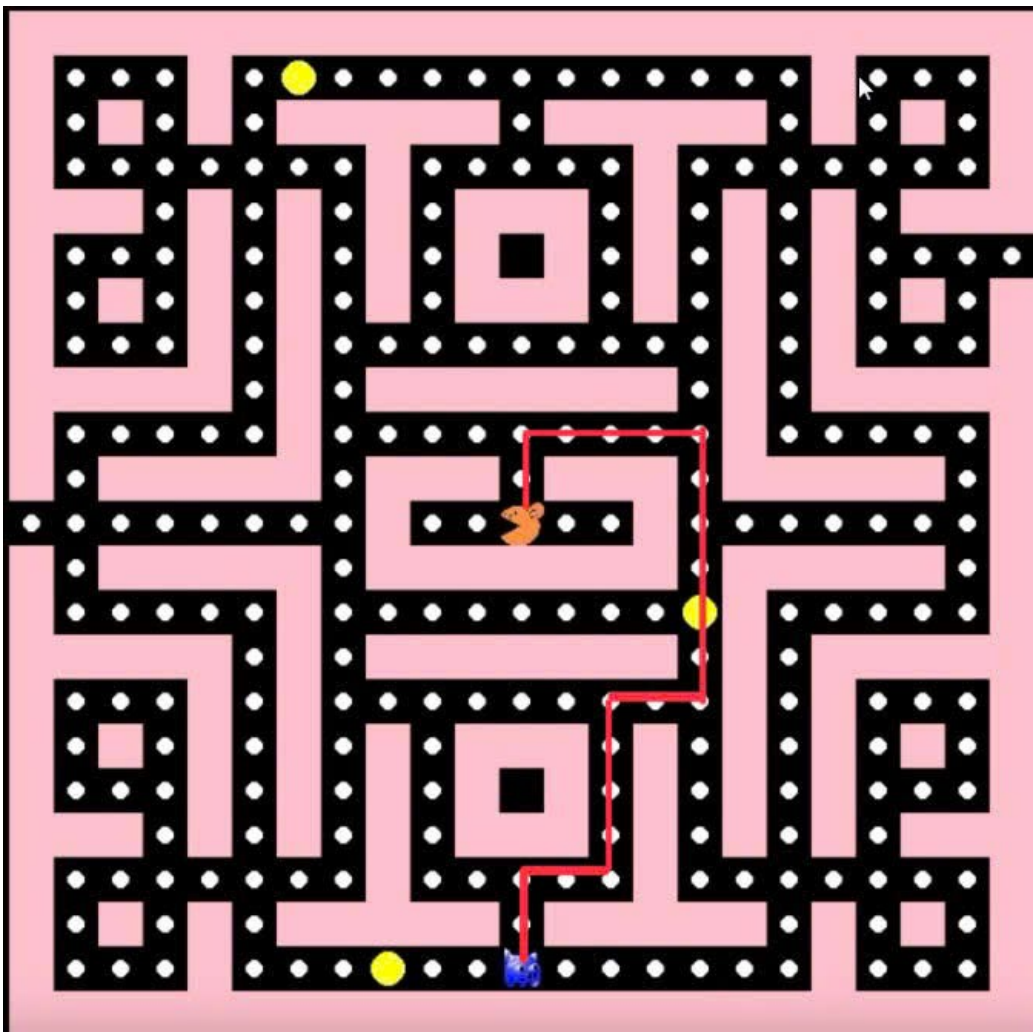

Figure 19: BFS

* **Search Time:** 0.001358747s

* **Memory Used:** 9168 bytes

* **Expanded Nodes:** 161

– **DFS**

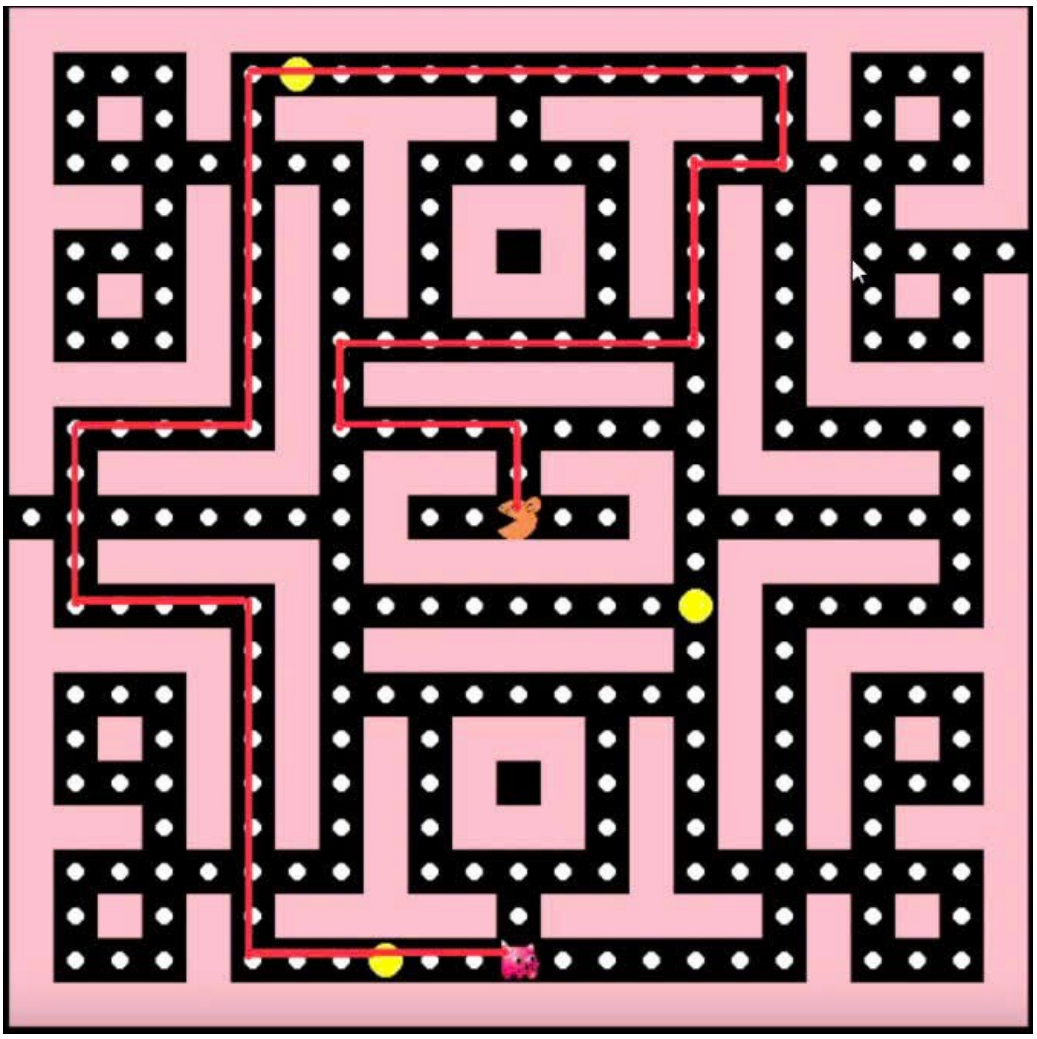

Figure 20: DFS

* **Search Time:** 0.011995554s

* **Memory Used:** 8656 bytes

* **Expanded Nodes:** 233

– **UCS**

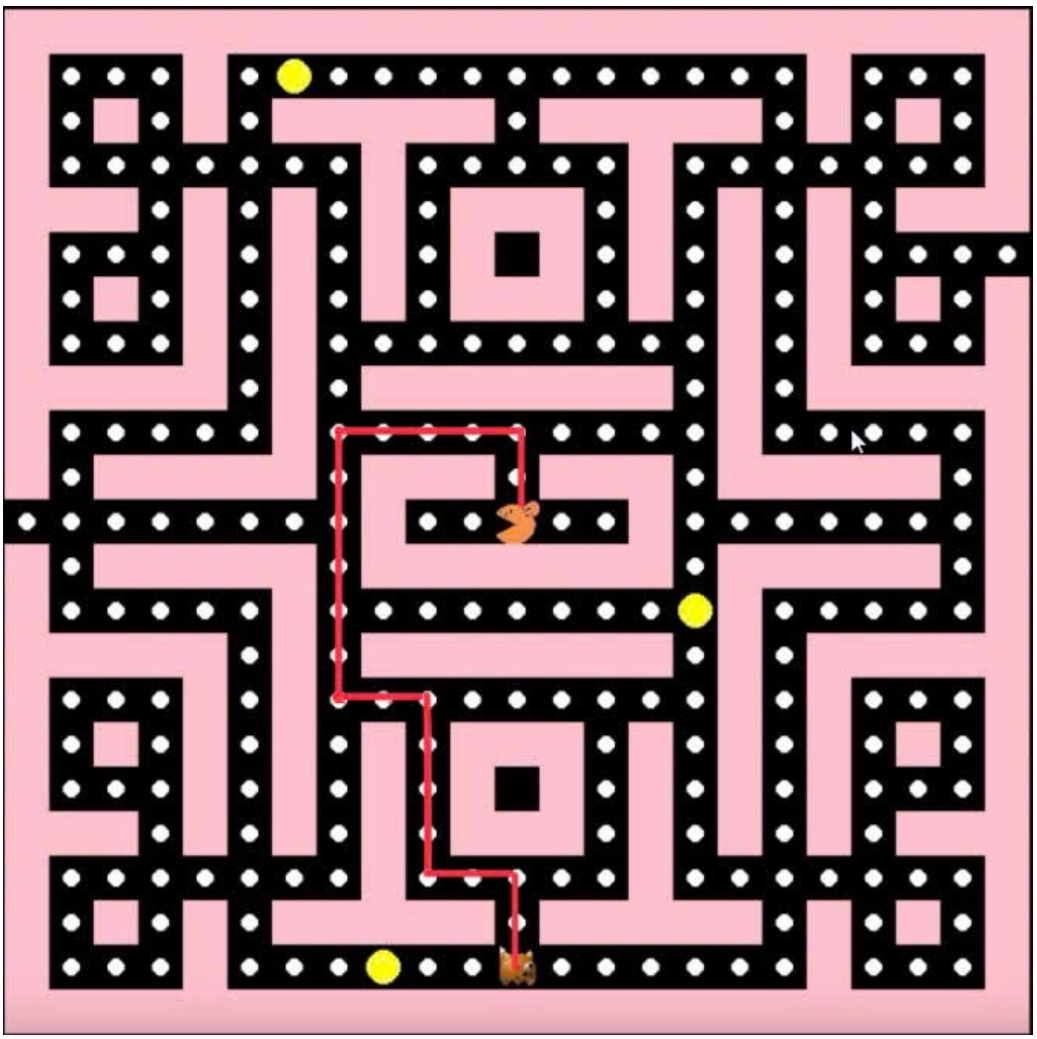

Figure 21: UCS

* **Search Time:** 0.000563383s

* **Memory Used:** 8624 bytes

* **Expanded Nodes:** 166

– **A***

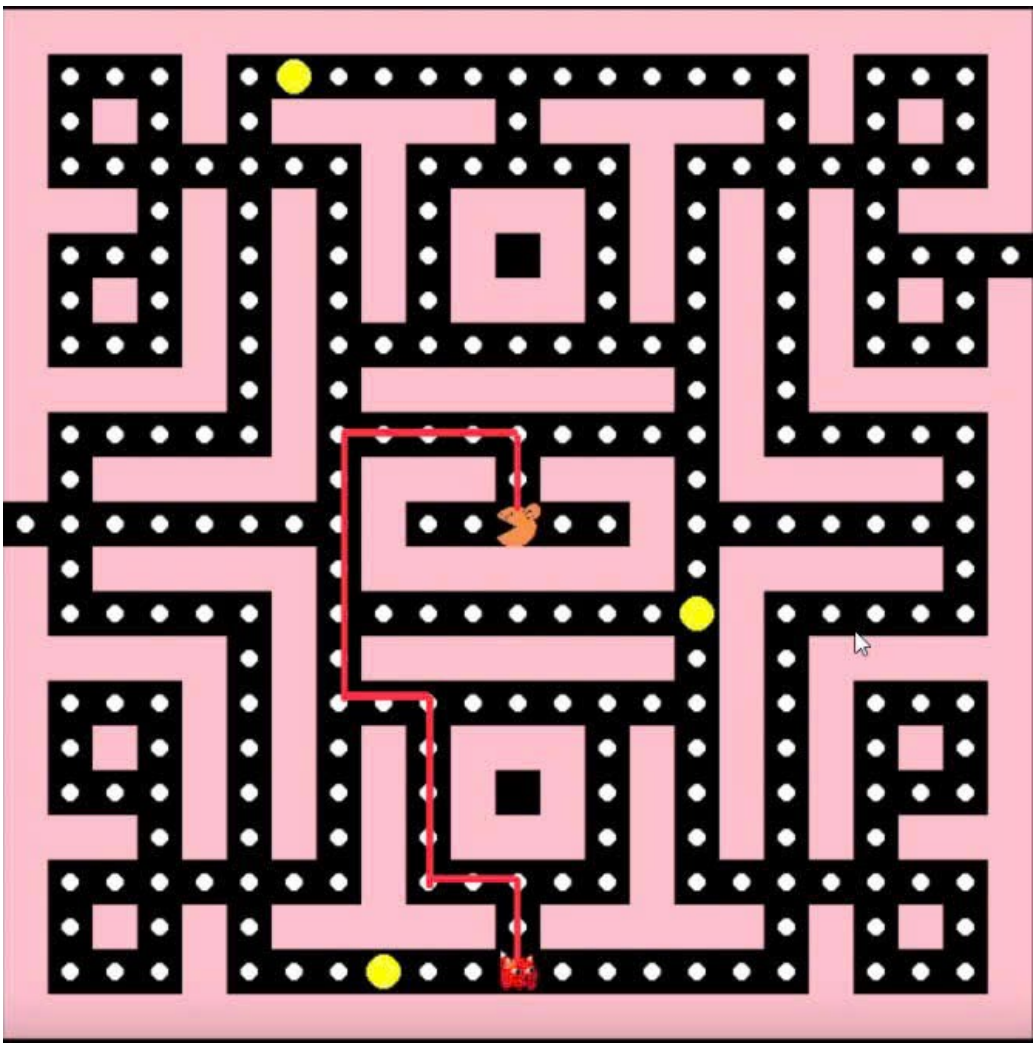

Figure 22: A*

  * **Search Time:** 0.000473738
  * **Memory Used:** 8422 bytes
  * **Expanded Nodes:** 78

- **Test case 5**
  - **Ghost Coordinate** (15, 8)
  - **Pacman Coordinate** (1, 7)
  - **BFS**

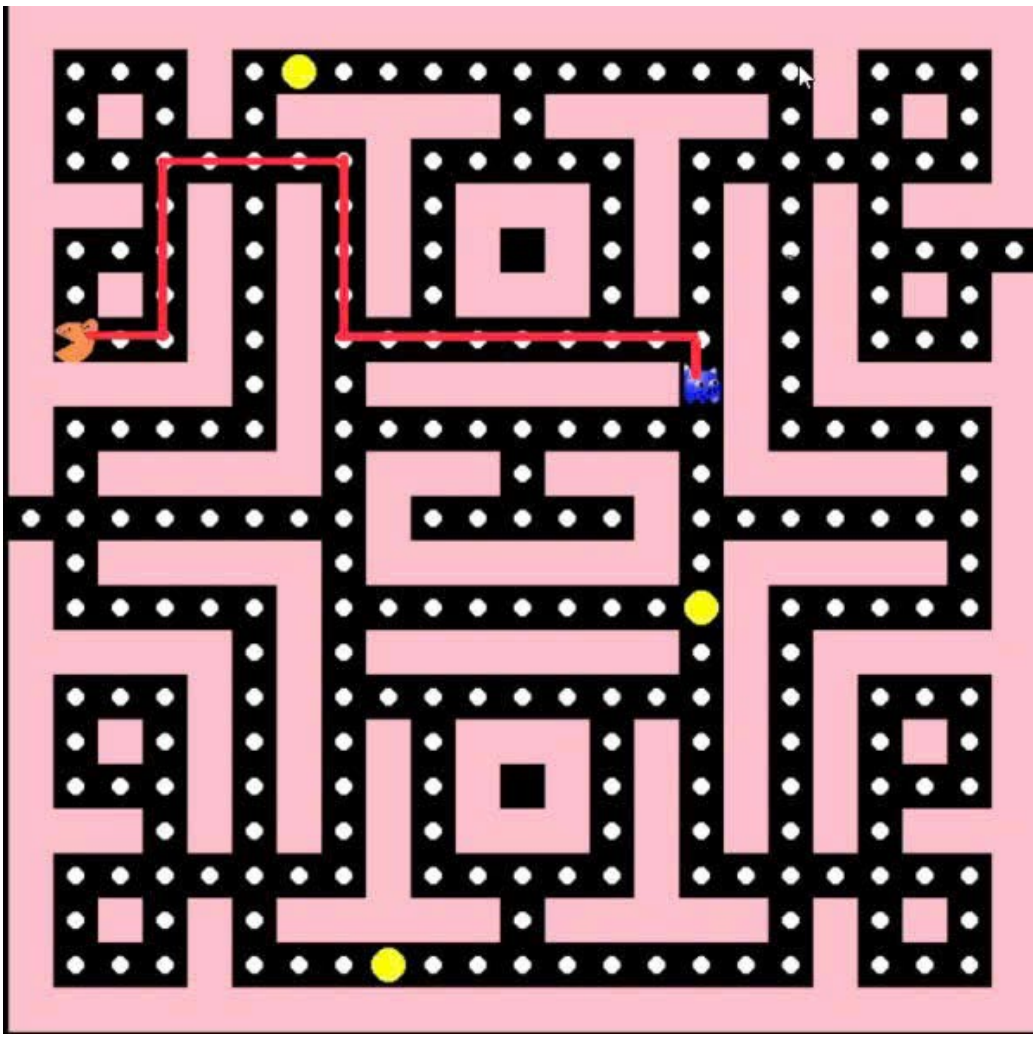

Figure 23: BFS

* **Search Time:** 0.000903606s

* **Memory Used:** 9168 bytes

* **Expanded Nodes:** 235

– **DFS**

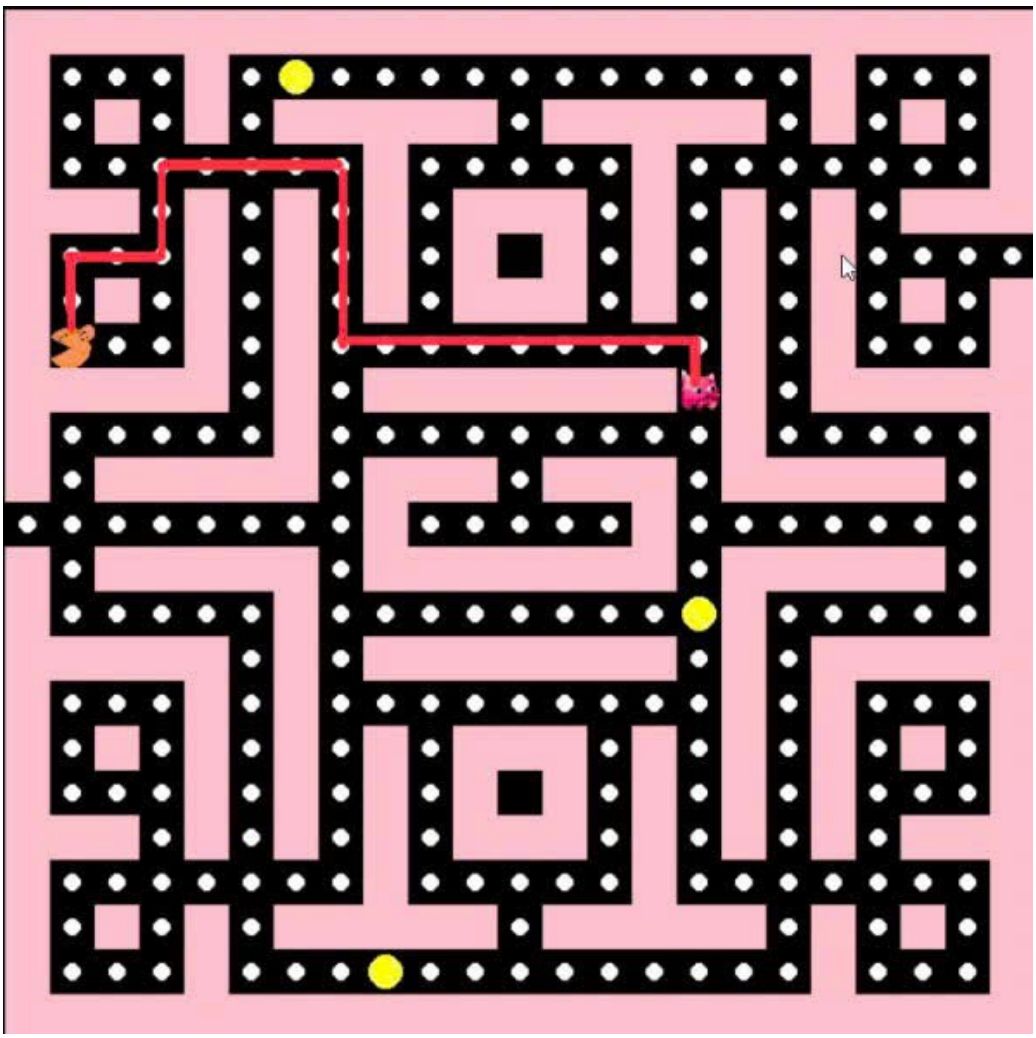

Figure 24: DFS

* **Search Time:** 0.000130414s

* **Memory Used:** 2448 bytes

* **Expanded Nodes:** 31

– **UCS**

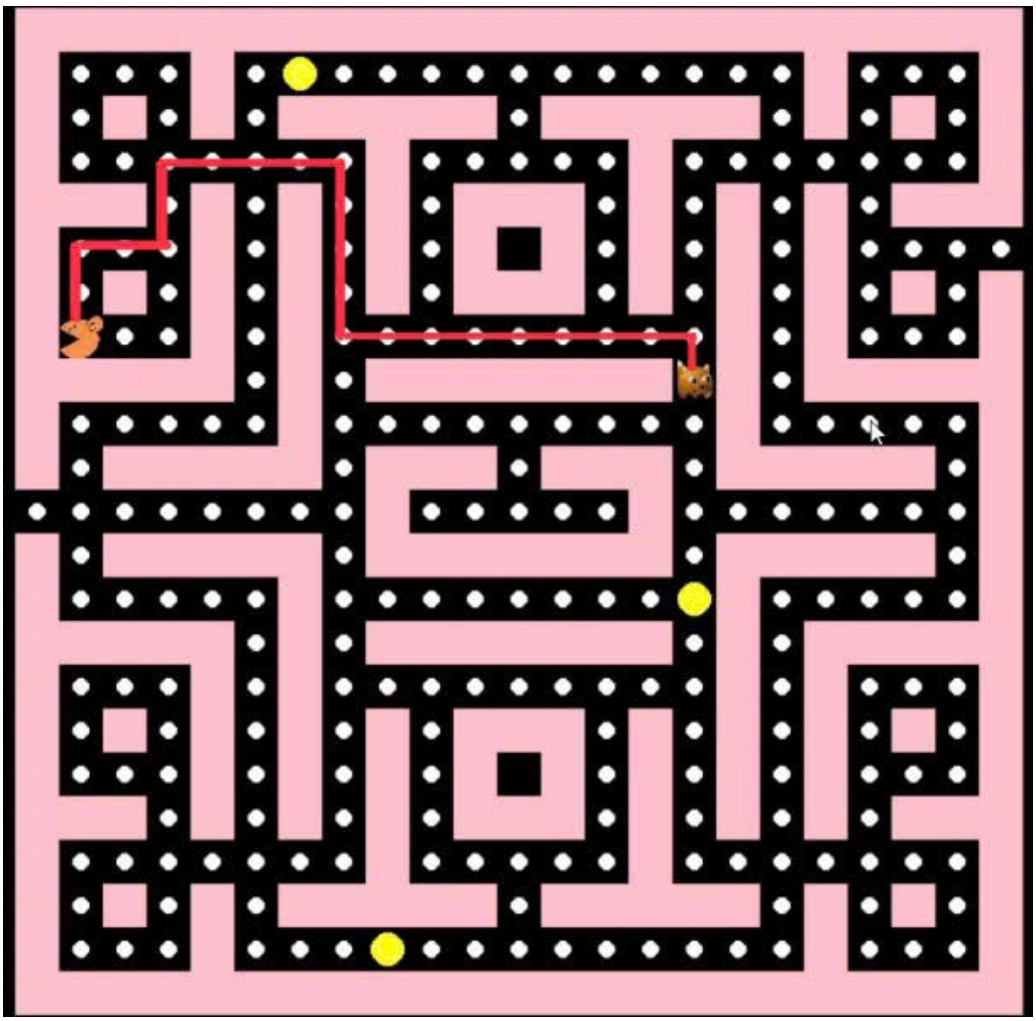

Figure 25: UCS

* **Search Time:** 0.001136065s

* **Memory Used:** 8656 bytes

* **Expanded Nodes:** 227

– **A***

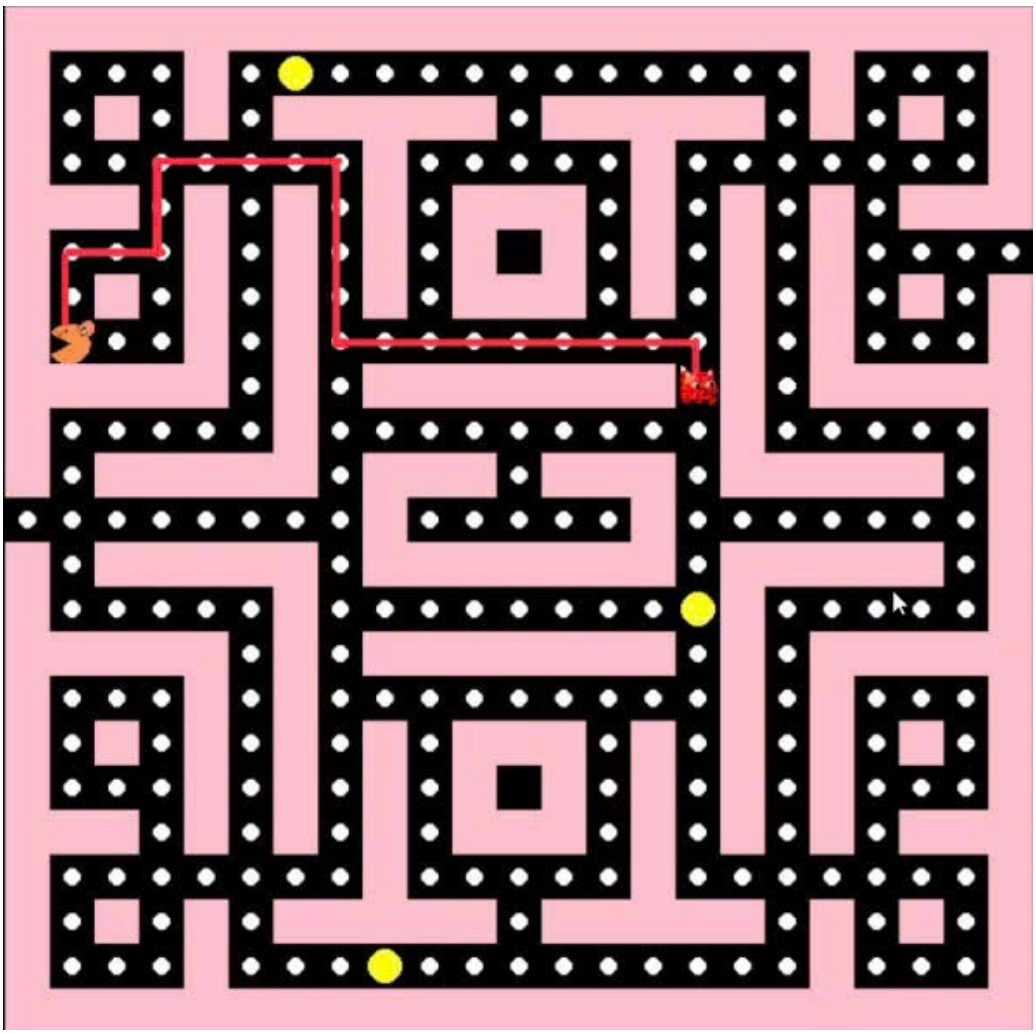

Figure 26: A*

* **Search Time:** 0.000451326s
* **Memory Used:** 8424 bytes
* **Expanded Nodes:** 77

4 Analyze and evaluate algorithms

Test case 1: Ghost (1,1) _ PacMan (21,21)			
Algorithm	Memory used	Searching times	Expanded nodes
BFS	9168 bytes	0.001940966 s	251
DFS	8720 bytes	0.001270 s	217
UCS	8624 bytes	0.000520706 s	255
A*	8424 bytes	0.001005650 s	41

Figure 27: Aggregated data from test case 1

Test case 2: Ghost (10, 1) _ PacMan (1, 15)			
Algorithm	Memory used	Searching times	Expanded nodes
BFS	9168 bytes	0.00143027305 s	257
DFS	8624 bytes	0.000892878 s	125
UCS	8592 bytes	0.000960588 s	249
A*	8424 bytes	0.000986099 s	140

Figure 28: Aggregated data from test case 2

Test case 3: Ghost (22, 5) _ PacMan (9, 11)			
Algorithm	Memory used	Searching times	Expanded nodes
BFS	9168 bytes	0.001305103 s	104
DFS	8656 bytes	0.011940956 s	256
UCS	8592 bytes	0.000392437 s	107
A*	2276 bytes	0.000247002 s	45

Figure 29: Aggregated data from test case 3

Test case 4: Ghost (11, 21) _ PacMan (11, 11)			
Algorithm	Memory used	Searching times	Expanded nodes
BFS	9168 bytes	0.001358747 s	161
DFS	8656 bytes	0.011995554 s	233
UCS	8624 bytes	0.000563383 s	166
A*	8422 bytes	0.000473738 s	78

Figure 30: aggregated data from test case 4

Test case 5: Ghost (15, 8) _ PacMan (1, 7)			
Algorithm	Memory used	Searching times	Expanded nodes
BFS	9168 bytes	0.000903606 s	235
DFS	2448 bytes	0.000130414 s	31
UCS	8656 bytes	0.001136065 s	227
A*	8424 bytes	0.000451326 s	77

Figure 31: aggregated data from test case 5

4.1 Breadth-First Search (BFS)

4.1.1 Analysis of Results

- **Memory Usage:** Highest among all algorithms (9168 bytes).
- **Search Time:** Generally fast but not always the fastest (ranging from 0.0009s to 0.0019s).
- **Expanded Nodes:** Very high (ranging from 104 to 257).

4.1.2 Explanation

BFS expands states layer by layer, meaning it explores all states at distance d before expanding states at $d+1$. It guarantees finding the shortest path in an unweighted graph but requires storing all expanded states, leading to high memory consumption. Due to this exhaustive exploration, BFS expands a large number of nodes.

4.2 Depth-First Search (DFS)

4.2.1 Analysis of Results

- **Memory Usage:** Varies significantly (from 2448 bytes to 8720 bytes).
- **Search Time:** Highly unstable (can be extremely fast: 0.0001s, or very slow: 0.0119s).
- **Expanded Nodes:** Varies widely (from 31 to 256).

4.2.2 Explanation

DFS explores the deepest branch first rather than searching layer by layer like BFS. If DFS happens to pick the correct path early, it can find the solution extremely fast. However, if it chooses the wrong direction, it may explore almost the entire search space before finding the solution, leading to long search times and high node expansions. DFS uses less memory than BFS since it only stores the current path.

4.3 Uniform Cost Search (UCS)

4.3.1 Analysis of Results

- **Memory Usage:** Lower than BFS but higher than DFS (typically 8592-8656 bytes).
- **Search Time:** Can be the fastest in some cases (e.g., 0.0003s) but sometimes slower.
- **Expanded Nodes:** High but not as high as BFS.

4.3.2 Explanation

UCS prioritizes expanding the node with the lowest cost first rather than expanding layer by layer like BFS. This ensures UCS finds the shortest path even when the graph has different edge weights. However, since UCS has no guidance toward the goal, it may expand many unnecessary nodes. It can be faster than BFS when edge weights favor an optimal path but may expand almost as many nodes as BFS in uniform-weight scenarios.

4.4 A* Algorithm

4.4.1 Analysis of Results

- **Memory Usage:** The lowest in most cases (8424 bytes, sometimes only 2276 bytes).
- **Search Time:** Always among the fastest (0.0002s to 0.001s).
- **Expanded Nodes:** The lowest (from 41 to 140).

4.4.2 Explanation

A* uses an evaluation function to prioritize nodes with the lowest cost plus an estimated distance to the goal. This allows A* to focus its search in the direction most likely to reach the goal quickly. Unlike BFS and UCS, A* expands significantly fewer nodes by selecting promising states. With a well-designed heuristic function, A* typically finds the optimal path while expanding the fewest nodes.

4.5 Summary Table

Algorithm	Memory Usage	Speed	Efficiency
BFS	Highest	Average	Good but resource-intensive
DFS	Variable, can be low	Unstable	Not guaranteed optimal
UCS	Medium	Variable	Good but may expand many nodes
A*	Lowest	Fastest	Most efficient

Table 3: Comparison of search algorithms based on memory, speed, and efficiency.

4.6 General Conclusion

Finding the fastest and most optimal path from Ghost to Pacman, **A*** is the best choice because:

- It finds the shortest path.
- It uses the least memory.
- It expands the fewest nodes.
- **If a good heuristic is not available, UCS is also a reasonable choice.**
- **BFS** is suitable to ensure the shortest path without concern for efficiency.
- **DFS** is not recommended as it does not guarantee a good path and may take excessive time.

5 Video Game

Video game: https://youtu.be/EolYpvMNnNs?si=N1V4Hm6ZqkXVKQgC

References

[1] "A* search algorithm." https://www.geeksforgeeks.org/a-search-algorithm/, 2024.

[2] "Pac-man." https://www.youtube.com/watch?v=9H27CimgPsQ&t=906s, 2023.

[3] "Pygame tutorial." https://www.geeksforgeeks.org/pygame-tutorial/, 2024.