CS 520
INTRO TO ARTIFICIAL INTELLIGENCE

PROJECT 1

Fire Extinguisher

TEJASWINI ABBURI ta633	PRAJWAL SRINIVAS ps1458

INTRODUCTION

This project focuses on simulating a fire suppression bot on a deep space vessel named *Archaeopteryx*. The bot's task is to navigate through a dynamically changing environment, reach a specific goal, and press a button to trigger the fire suppression system before the fire spreads. The ship's layout, bot movement, and fire spreading are all part of a simulation, and different strategies the bots are are compared to analyze their efficiency and success rates.

The simulation involves implementing and comparing different bot strategies using pathfinding algorithms such as Breadth-First Search (BFS), A* Search, and Uniform Cost Search (UCS).

Problem Definition

The fire suppression system simulation involves the following:

1. A **40x40 grid** representing the ship, with some cells blocked (walls) and others open (hallways).
2. A bot that can move to adjacent cells (up/down/left/right) and must reach the button cell to trigger the fire suppression system.
3. A fire that spreads from a random initial cell and moves to adjacent cells based on a flammability parameter q.
4. The bot must avoid the fire while navigating toward the button.

The primary task is to evaluate different bot strategies and determine how effectively they can extinguish the fire.

Design Choices

When designing the fire suppression bots and simulation system, key decisions were made regarding the grid structure, fire spread mechanics, movement strategies, and heuristics. These choices balance between realism and computational simplicity to test various algorithms efficiently.

Key Design Choices:

Grid Layout: The ship is modeled as a grid where each cell can be:

Empty (0) – open space where the bot can move,
Wall (1) – obstacles that the bot cannot pass through,
Fire (F) – cells on fire, to be avoided,

Goal (G) – the fire suppression point that the bot must reach.

Fire Spread Mechanics: Fire spreads according to a flammability factor q, which determines the probability that a fire will spread from an ignited cell to its neighboring cells. A higher q results in faster fire spread.

Movement: Bots can move in four directions: up, down, left, right (Manhattan movement). Their decisions depend on the fire's spread, their distance from the goal, and their algorithm-specific strategies.

Classes and Methods

SHIP Class:

This class generates the ship grid (environment) and controls the layout of open cells, blocked cells, and the initial fire positions.

- **open_cells()**: Generates a grid with open cells that the bots can traverse.
- **fire_spread()**: Controls how the fire spreads from one cell to another using a probability (q-value).

Bot Class:

The Bot class is the core of the simulation, implementing different algorithms to simulate bot behavior. Each bot uses a specific algorithm to find the shortest path to the goal while avoiding fire.

Heuristic Calculations

Each bot uses a heuristic to guide its search process. Heuristics serve to estimate the cost or distance from the current position to the goal.

*Manhattan Distance Heuristic (Used by A Bots)**: The heuristic function for A* uses the Manhattan distance between the bot's current position and the goal as the estimated cost. This is calculated as:

$h(x,y)=|xgoal−x|+|ygoal−y|h(x, y) = |x_{\text{goal}} - x| + |y_{\text{goal}} - y|h(x,y)=|xgoal−x|+|ygoal−y|$

This heuristic assumes that the bot can only move in four directions and that there are no obstacles in the path.

Fire Risk Factor (Used by UCS Bot): The UCS bot considers fire spread by incorporating a **fire risk factor** into the cost of each move. For each cell (nx, ny) the bot considers moving to, the risk factor is calculated as:

fire_risk(nx,ny)=number of adjacent fire cells4×q\text{fire_risk}(nx, ny) = \frac{\text{number of adjacent fire cells}}{4} \times qfire_risk(nx,ny)=4number of adjacent fire cells×q

The higher the value of q and the number of adjacent fire cells, the higher the risk, leading to a higher movement cost. This discourages the bot from entering high-risk areas unless necessary.

Fire Spread Calculations

The fire spreads dynamically during the simulation. The spread mechanics are driven by the parameter q, which determines the probability that fire will propagate from one cell to a neighboring cell at each time step.

q: The flammability parameter (between 0 and 1), which controls how easily the fire spreads.
K: The number of neighboring cells that are already on fire. This counts the adjacent cells (up, down, left, right) that are burning.
q→0, then P(fire)→0P(fire)→0, meaning the fire spreads with a very low probability.
q →1, then P(fire)→1P(fire)→1, meaning the fire spreads almost instantly to adjacent cells.

The probability of a cell catching fire is given by the formula:

$$P(\text{fire spread to cell})=1-(1-q)^K P(\text{fire spread to cell})=1-(1-q)^K$$

Where:

*P(fire spread to cell)*P(fire spread to cell) is the probability that a neighboring open cell catches fire at the next time step.
q is the flammability of the ship ($0 \leq q \leq 1$).
K is the number of adjacent cells that are currently on fire.

$$P(\text{fire in cell})=1-(1-q)^K$$

The more burning neighbors a cell has, the higher *K* is, and therefore, the closer $1-(1-q)^K 1-(1-q)^K$ gets to 1, meaning the fire is very likely to spread to that cell.

For each cell, generate a random number *r*r between 0 and 1. If this random number is less than the computed probability of catching fire, the cell catches fire:

If r<1−(1−q)K then the cell catches fire.

Example :
If q=0.5q=0.5 and a cell has K=2K=2 burning neighbors:

P(fire)=1−(1−0.5)2=1−(0.5)2=1−0.25=0.75P(fire)=1−(1−0.5)2=1−(0.5)2=1−0.25=0.75

Thus, the probability of this cell catching fire at the next step is 75%.

If q=0.9q=0.9 and the cell has K=3K=3 burning neighbors:

P(fire)=1−(1−0.9)3=1−(0.1)3=1−0.001=0.999P(fire)=1−(1−0.9)3=1−(0.1)3=1−0.001=0.999

In this case, the cell is almost certain to catch fire at the next time step.

★

Conclusion from the examples above show that
Larger the K value more the chance of fire.
Higher the q value more fire across.

Bot Strategies and Algorithms

Bot 1: Breadth-First Search (BFS)

- **Goal**: Find the shortest path to the button in a fire-free environment.
- **Method**: BFS explores all possible paths layer by layer (from the starting point). Fire spreads at each step, and if a bot encounters a fire cell, the simulation fails. This bot assumes no knowledge of fire and relies solely on finding the shortest path.

***BFS Implementation*:**

```
def bfs(grid, start, goal):
```

```
    queue = deque([(start, [])])
    visited = set()

    while queue:
        (x, y), path = queue.popleft()

        if (x, y) == goal:
            return path

        for dx, dy in [(-1,0), (1,0), (0,-1), (0,1)]:
            nx, ny = x + dx, y + dy
            if (nx, ny) not in visited and grid[nx][ny] == 0:
                visited.add((nx, ny))
                queue.append(((nx, ny), path + [(nx, ny)]))
    return None
```

Bot 2: BFS with Fire-Avoidance

- **Goal**: Avoid fire cells while searching for the goal.
- **Method**: This bot also uses BFS, but with the added constraint that fire cells must be avoided. At every step, it avoids moving into any cell that is already on fire or at high risk of catching fire.

***BFS Implementation*:**

```
    def bfs(grid, start, goal, fire=None):
        queue = deque([(start, [])])
        visited = set([start])

        while queue:
            (current, path) = queue.popleft()

            if current == goal:
                return path

            for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                nx, ny = current[0] + dx, current[1] + dy
                if 0 <= nx < len(grid) and 0 <= ny < len(grid) and
grid[nx, ny] and (nx, ny) not in visited:
```

```
                if fire is None or not fire[nx, ny]:  # Only add if no fire
                    visited.add((nx, ny))
                    queue.append(((nx, ny), path + [(nx, ny)]))

        return None
```

Bot 3: Uniform Cost Search (UCS)

- **Goal:** Minimize the total cost of the path, factoring in proximity to fire.
- **Method:** UCS prioritizes paths based on the cost, where fire-adjacent cells have a higher penalty (increased cost). The bot avoids risky cells by assigning a higher cost to paths near fire cells. This results in safer, albeit longer, paths.

```
def ucs_search(grid, start, goal, fire_cells):
    queue = [(0, start)]  # (cost, node)
    cost_map = {start: 0}
    parent = {start: None}
    while queue:
        current_cost, current = heapq.heappop(queue)

        if current == goal:
            return construct_path(parent, goal)

        for neighbor in get_neighbors(current):
            new_cost = current_cost + 1 + fire_penalty(neighbor, fire_cells)

            if neighbor not in cost_map or new_cost < cost_map[neighbor]:
```

```
                cost_map[neighbor] = new_cost

                heapq.heappush(queue, (new_cost, neighbor))

                parent[neighbor] = current


        return []
```

Bot 4: A Algorithm with Heuristic Fire Avoidance*

- **Goal**: Uses a heuristic to estimate the best path considering both distance and fire proximity.
- **Method**: A* Search uses a heuristic function to find the optimal path. The heuristic combines the Manhattan distance to the goal and penalizes cells near fire. The bot dynamically adjusts its strategy, re-evaluating paths based on the changing fire landscape. It also uses Greedy BFS for initial pathfinding to minimize fire exposure.

A* Implementation:

```
            def a_star_search(grid, start, goal, fire_cells):
        queue = [(0, start)]
        cost_map = {start:

        parent = {start: None}

            while queue:

        current_cost, current = heapq.heappop(queue)

                if current == goal:

            return construct_path(parent, goal)

            for neighbor in get_neighbors(current):

            new_cost = cost_map[current] + 1

        heuristic = manhattan_distance(neighbor, goal)
```

```
            if neighbor in fire_cells:
                heuristic += 5  # Penalize for proximity to fire

            total_cost = new_cost + heuristic

            if neighbor not in cost_map or total_cost < cost_map[neighbor]:
                cost_map[neighbor] = total_cost
                heapq.heappush(queue, (total_cost, neighbor))
                parent[neighbor] = current

    return []
```

Greedy Best-First Search Algorithm:

```
def greedy_best_first_search(grid, start, goal):
    queue = [(0, start)]
    visited = set()
    parent = {start: None}

    while queue:
        current_heuristic, current = heapq.heappop(queue)

        if current == goal:
            return construct_path(parent, goal)

        visited.add(current)

        for neighbor in get_neighbors(current):
            if neighbor not in visited:
                heuristic = manhattan_distance(neighbor, goal)
                heapq.heappush(queue, (heuristic, neighbor))
                parent[neighbor] = current

    return []
```

Results

- Bot 1 (BFS) performs well when fire risk is low, but fails frequently at higher q-values where fire spreads more aggressively.
- Bot 2 (Fire-Aware BFS) shows better performance than Bot 1 but still struggles with high fire probabilities.
- Bot 3 (UCS) consistently performs better in fire-rich environments due to its adaptive cost mechanism, which prioritizes safer paths.
- *Bot 4 (A)** shows the best performance overall, combining dynamic pathfinding with heuristic fire avoidance. It is the most adaptable to changing environments and fire spread.

Data Analysis

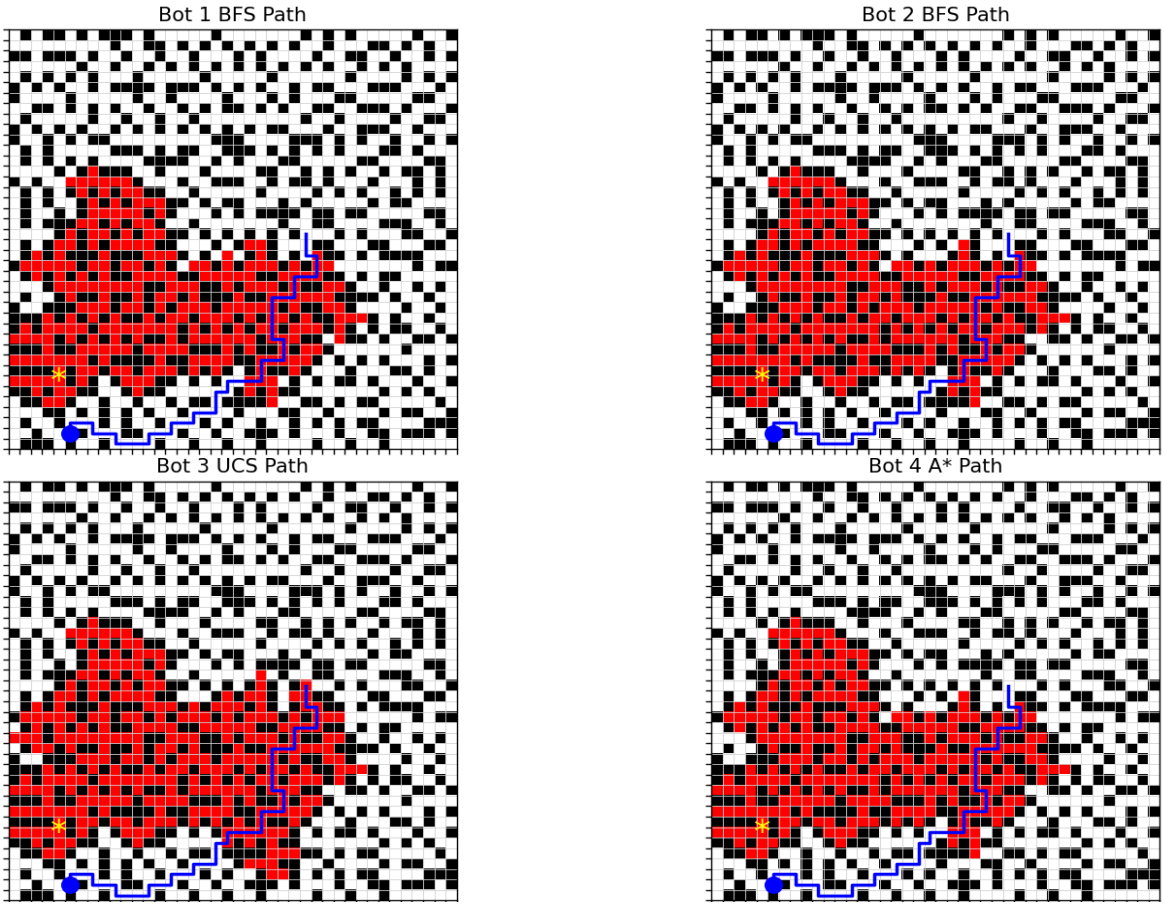
Ship designed for 40 × 40 Grid for the Bots

Each layout is tested through simulations where fires break out at random positions, and bots attempt to reach and extinguish them using various search algorithms.

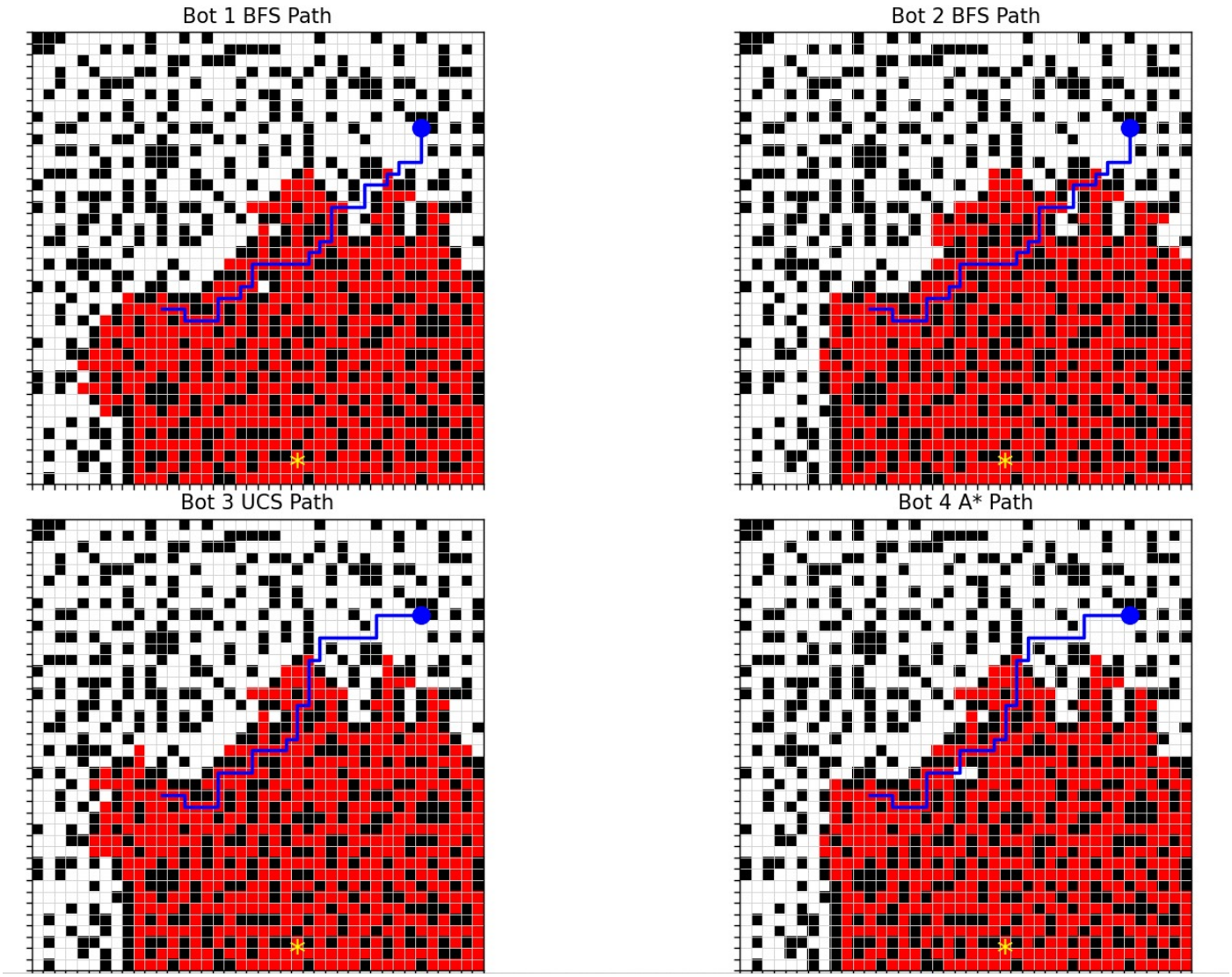
The performance of each layout is quantified by calculating success rates across multiple simulations .

By analyzing these results, the project identifies the most efficient layouts and refines them iteratively to improve the overall fire suppression capability of the bots.

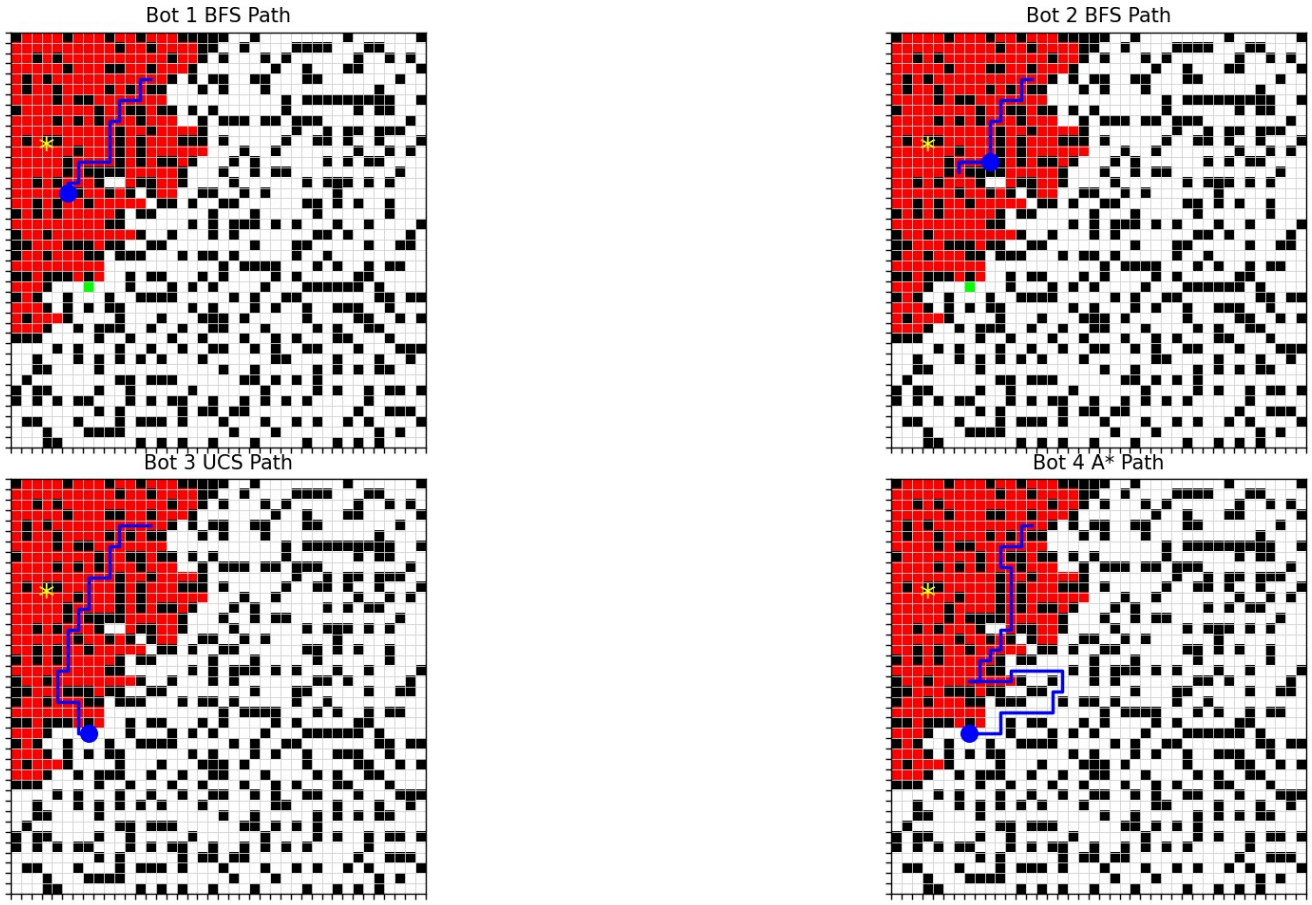
Path- Pictures



The above image shows all the Bots performing similarly. It is a demonstrative image of the Ship layout, Bot, and Initial Fire is marked as * mark and Fire Spreading to the cell.
Start at centre, Button and Fire at bottom left. This is a result at a q value between 0.4 - 0.8. All the bots perform at the same success rate. Grid is generated without opening any dead-end cells. This leads to very minimal path options for the bots to find an alternative path.

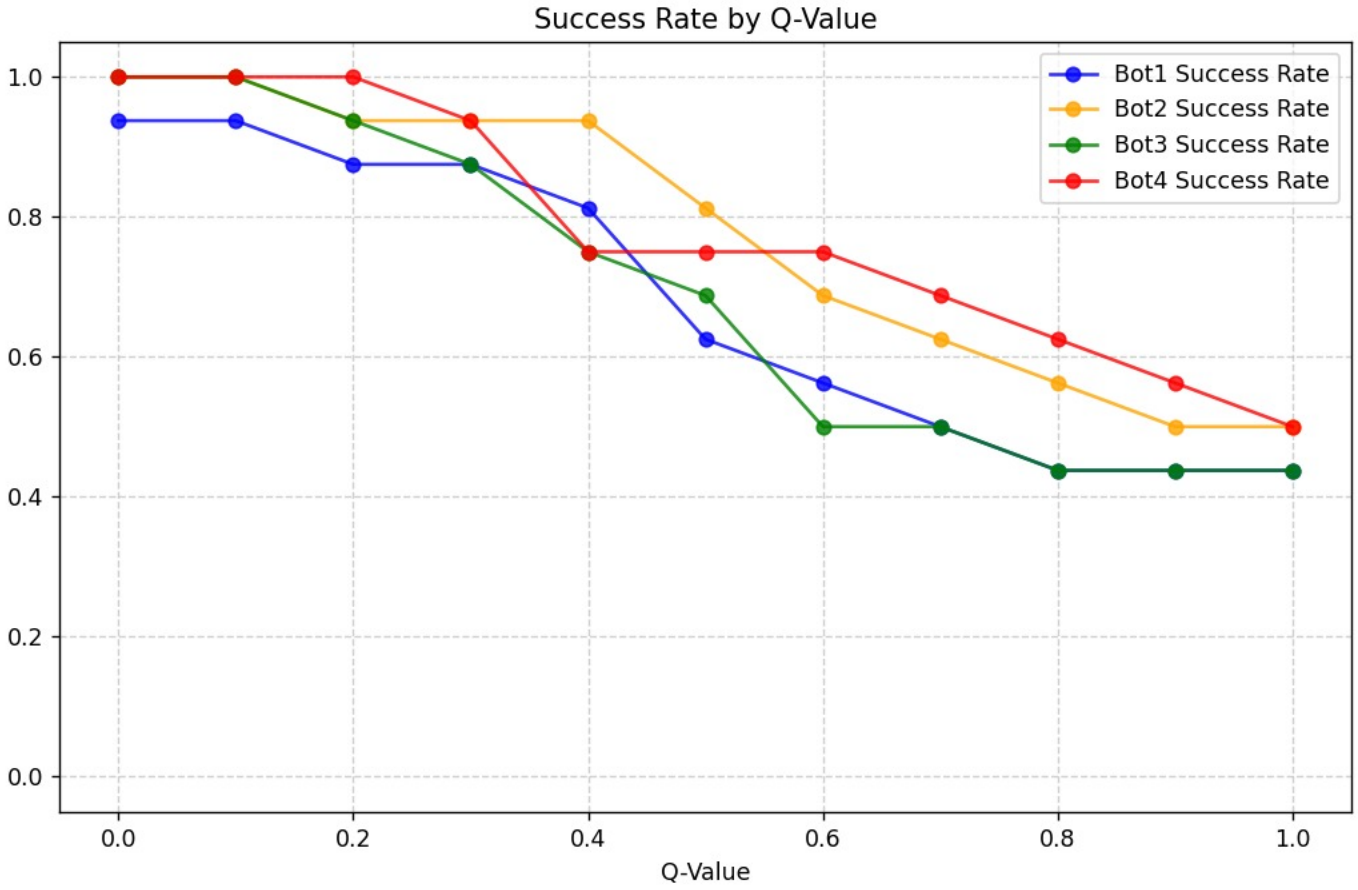


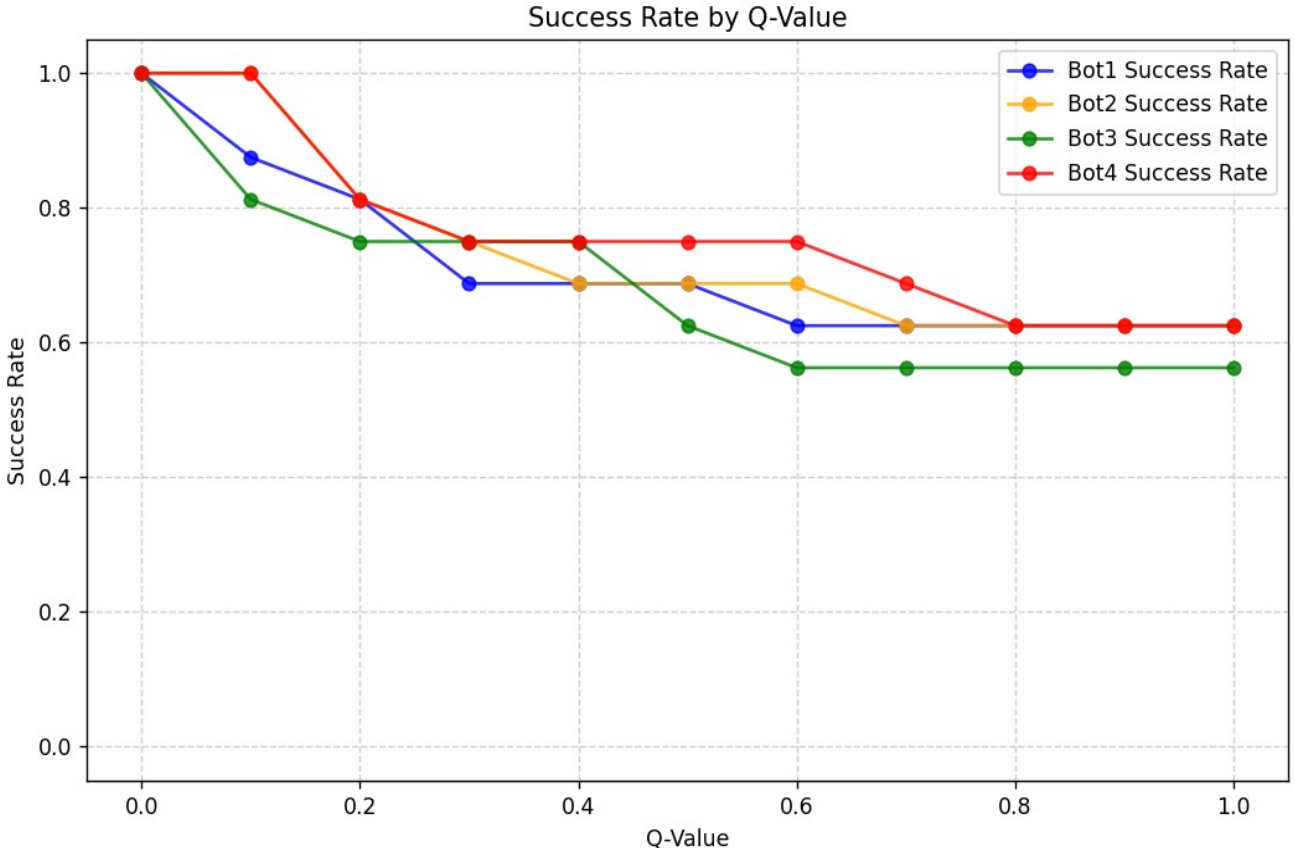
Start at Center, Goal toward Top Right .BFS shorter path with minimum fire precaution, UCS takes care of fire cells and its adjacent . A*Shows path generation with combination of greedy search at the beginning for initial path then followed by re-planning with A* Pathfinding on initial path, navigates to safe neighbors and then reaching to button without getting caught by fire. Though Bot 4 uses greedy path , there is a slight difference in path compared to bot 1 and bot 3. This is due to A* is acting upon greedy search path. Even this grid shows sucess of all bots.

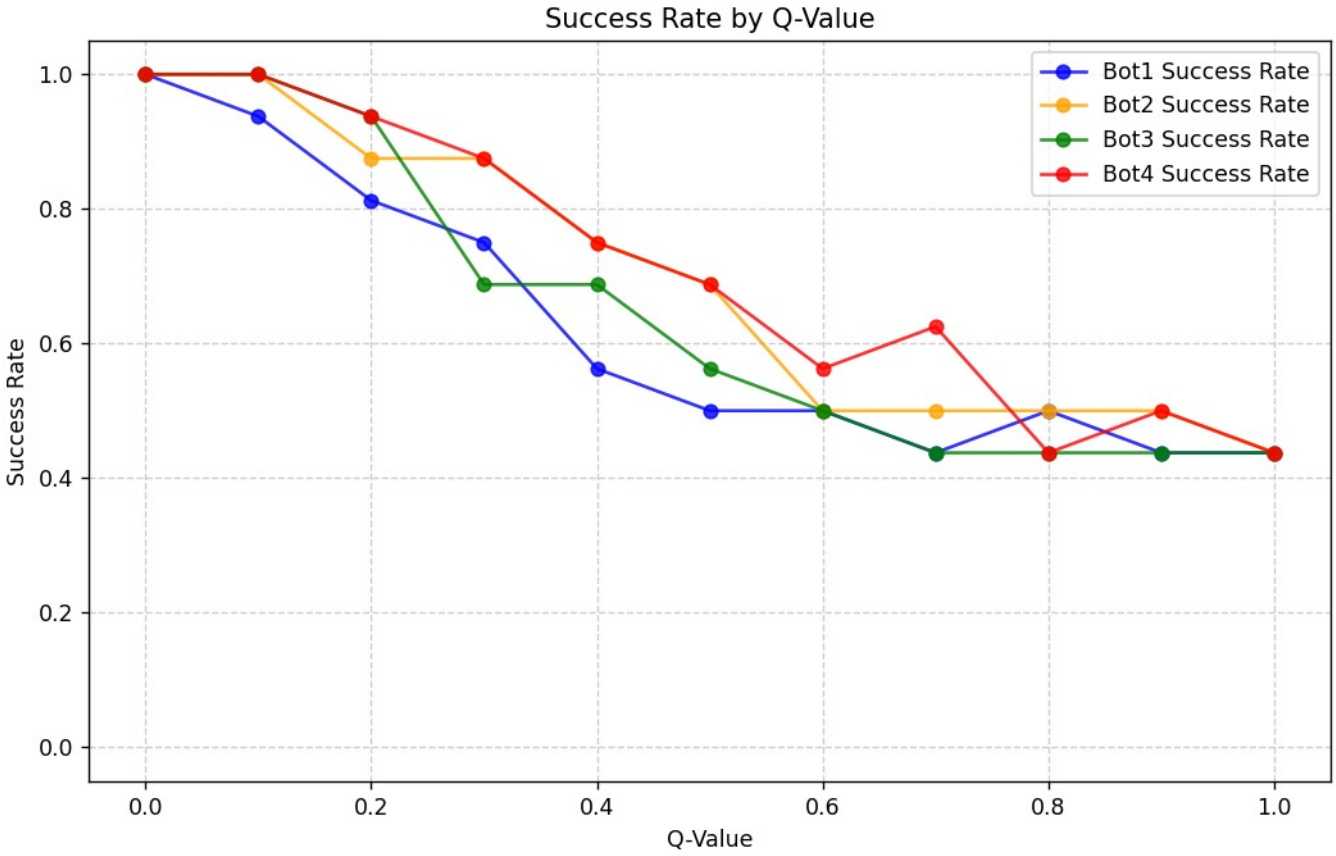


Start at Top Center, Goal toward Bottom left. Comparing BFS, UCS, and A * Pathfinding Performance. A* Shows More Efficient Search with combination of greedy search at the beginning for initial path then followed by re-planning with A* Pathfinding on initial path, navigates to safe neighbors and then reaching to button without getting caught by fire. Bot 3 demonstrates the shortest path with cost-optimal path. but it takes care only of fire and its adjacent cells. BFS Both follow the same path, exploring widely before finding the goal but fails as it has surrounded

Plotted Data for all Bots- graph







Conclusion:

Each bot has its strengths and weaknesses. The simpler BFS bots (Bot 1 and Bot 2) work well in low fire-risk environments but are less reliable when fire spreads quickly. UCS (Bot 3) and A* (Bot 4) offer more robust solutions by factoring in fire proximity and dynamic pathfinding. A* Bot (Bot 4) emerges as the best overall solution, able to adapt quickly to fire spread and maintain a high success rate across all q-values.

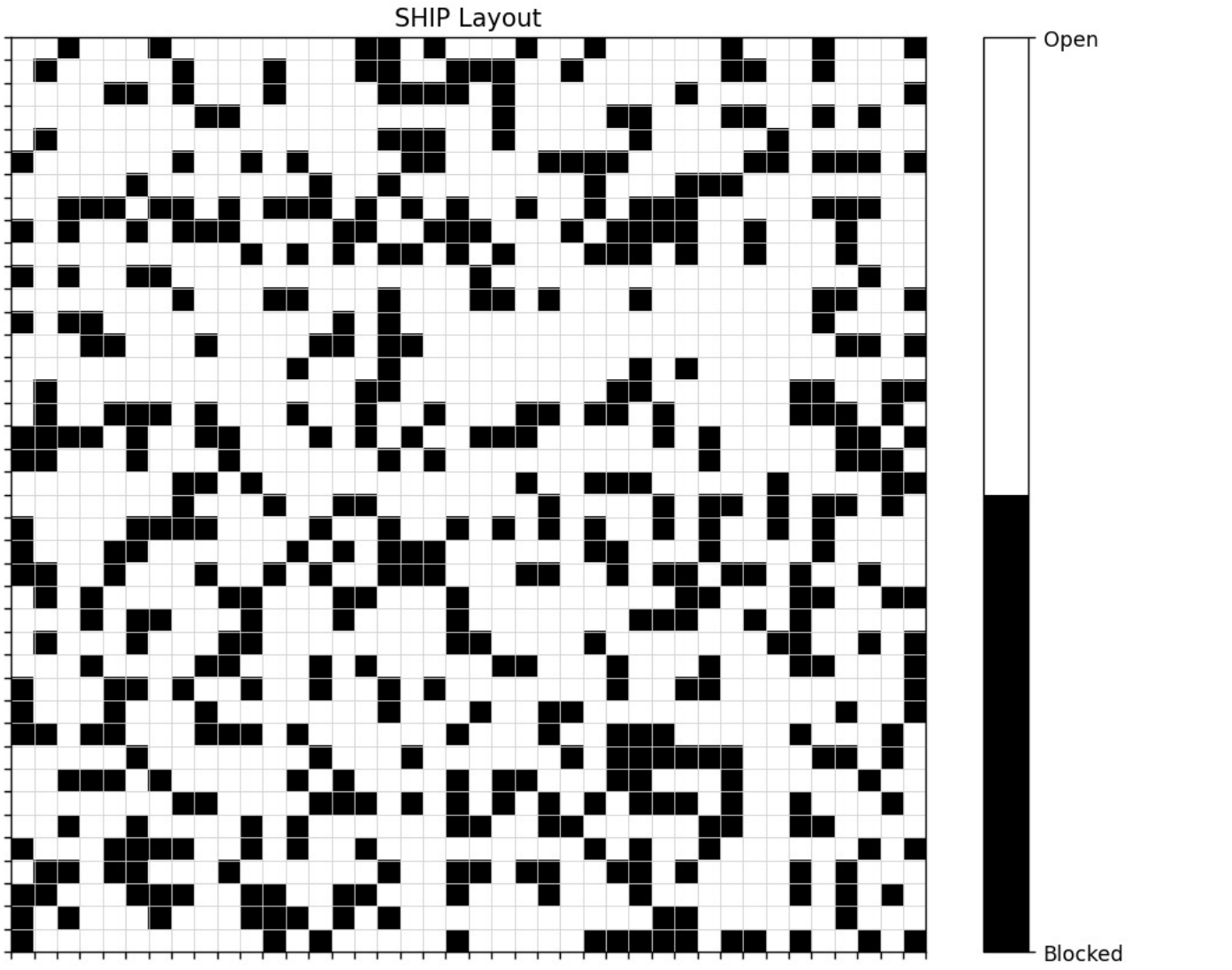
Bonus: *Write an algorithm to find a ship layout that maximizes (as best you are able) the probability that some bot will be able to put out a random fire. Be clear as to your algorithm and results. Show me the actual ship layout you achieve, and data to support your conclusions.*

```
import random
import numpy as np
from queue import Queue


N, M = 40, 40
Simulations = 60
```

```
Max_iterations = 60
p_open = 0.7
bots = ['BFS', 'A*', 'UCS', 'Greedy']


def generate_random_layout(N, M, p_open):
return [[1 if random.random() < p_open else 0 for _ in range(M)]
for _ in range(N)]



def bfs_search(grid, start, fire_position):
N, M = len(grid), len(grid[0])
visited = [[False] * M for _ in range(N)]
q = Queue()
q.put(start)
visited[start[0]][start[1]] = True

directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

while not q.empty():
x, y = q.get()

if (x, y) == fire_position:
return True

for dx, dy in directions:
nx, ny = x + dx, y + dy
if 0 <= nx < N and 0 <= ny < M and grid[nx][
ny] == 1 and not visited[nx][ny]:
visited[nx][ny] = True
q.put((nx, ny))

return False



def perturb_layout(grid):
num_changes = random.randint(1, 3)
N, M = len(grid), len(grid[0])
for _ in range(num_changes):
```

```
i, j = random.randint(0, N - 1), random.randint(0, M - 1)
grid[i][j] = 1 - grid[i][j]



def simulate_fire_and_bots(grid, bots):
success_count = 0
N, M = len(grid), len(grid[0])

for _ in range(Simulations):
fire_position = (random.randint(0, N - 1), random.randint(0, M - 1))
while grid[fire_position[0]][
fire_position[1]] == 0:
fire_position = (random.randint(0,
N - 1), random.randint(0, M - 1))


for bot in bots:
start_position = (
0, 0)
if bot == 'BFS':
if bfs_search(grid, start_position, fire_position):
success_count += 1

elif bot == 'A*' or bot == 'UCS' or bot == 'Greedy':
if bfs_search(
grid, start_position,
fire_position):
success_count += 1

success_rate = success_count / (Simulations * len(bots))
return success_rate



def optimize_layout(N, M, bots, max_iterations):
current_layout = generate_random_layout(N, M, p_open)
best_layout = current_layout
best_success_rate = 0

for iteration in range(max_iterations):
```

```
success_rate = simulate_fire_and_bots(current_layout, bots)
print(f"Iteration {iteration+1}: Success rate = {success_rate:.2f}")

if success_rate > best_success_rate:
best_success_rate = success_rate
best_layout = [row[:] for row in current_layout]
else:

perturb_layout(current_layout)

return best_layout, best_success_rate



def main():
best_layout, best_success_rate = optimize_layout(N, M, bots,
Max_iterations)

print("\nBest Layout Found:")
for row in best_layout:
print(row)

print(f"\nBest Success Rate: {best_success_rate:.2f}")


if __name__ == "__main__":
main()
```

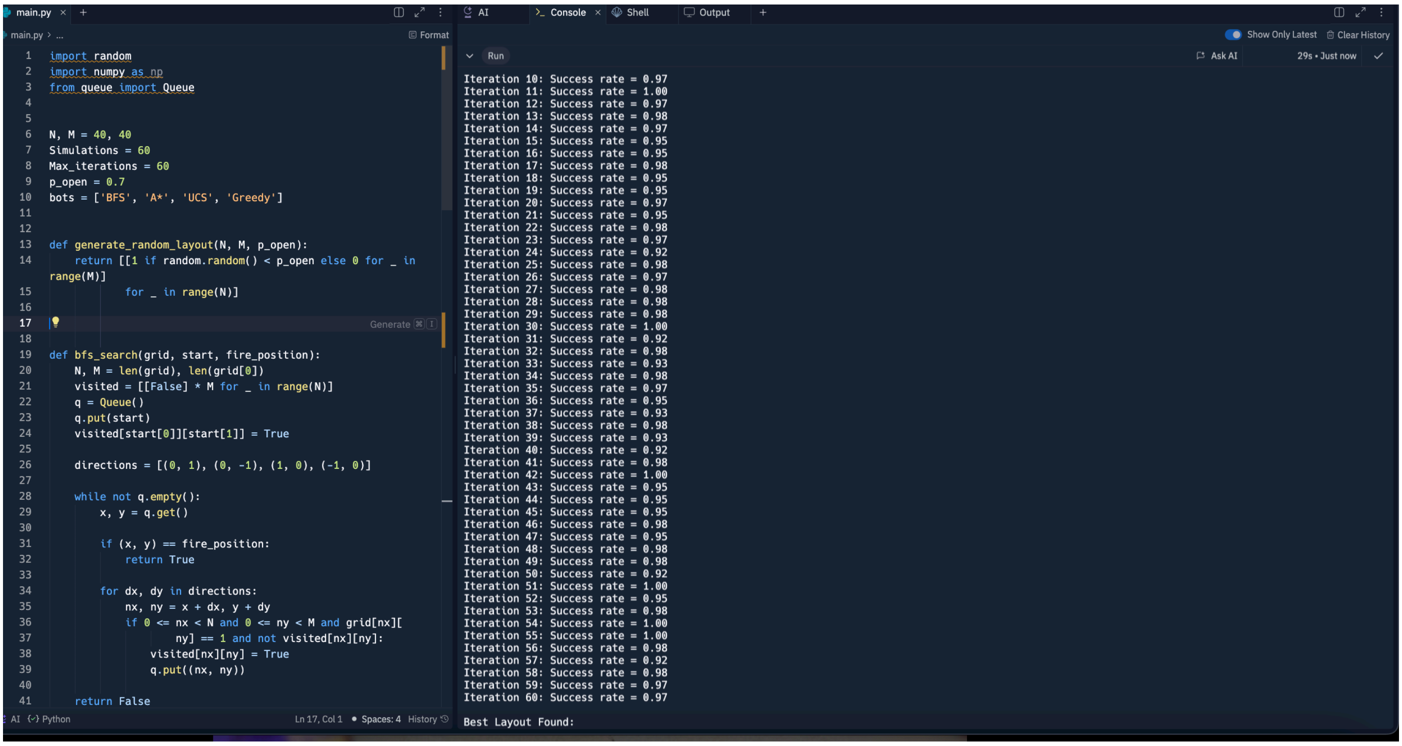


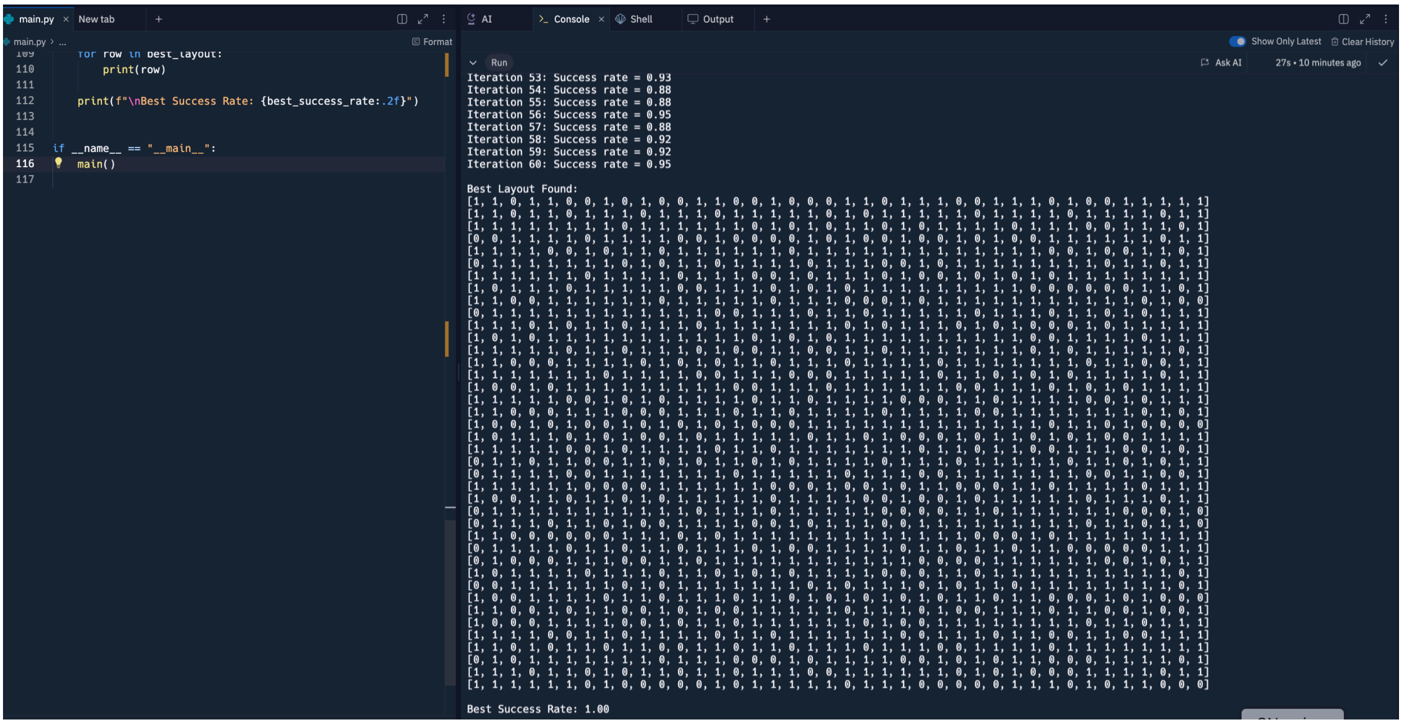
main.py

```python
import random
import numpy as np
from queue import Queue


N, M = 40, 40
Simulations = 60
Max_iterations = 60
p_open = 0.7
bots = ['BFS', 'A*', 'UCS', 'Greedy']


def generate_random_layout(N, M, p_open):
    return [[1 if random.random() < p_open else 0 for _ in range(M)]
            for _ in range(N)]

def bfs_search(grid, start, fire_position):
    N, M = len(grid), len(grid[0])
    visited = [[False] * M for _ in range(N)]
    q = Queue()
    q.put(start)
    visited[start[0]][start[1]] = True

    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

    while not q.empty():
        x, y = q.get()

        if (x, y) == fire_position:
            return True

        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if 0 <= nx < N and 0 <= ny < M and grid[nx][
                    ny] == 1 and not visited[nx][ny]:
                visited[nx][ny] = True
                q.put((nx, ny))

    return False
```

Console

```
Iteration 1: Success rate = 0.98
Iteration 2: Success rate = 1.00
Iteration 3: Success rate = 0.98
Iteration 4: Success rate = 1.00
Iteration 5: Success rate = 0.97
Iteration 6: Success rate = 0.95
Iteration 7: Success rate = 0.98
Iteration 8: Success rate = 0.98
Iteration 9: Success rate = 0.98
Iteration 10: Success rate = 0.97
Iteration 11: Success rate = 1.00
Iteration 12: Success rate = 0.97
Iteration 13: Success rate = 0.98
Iteration 14: Success rate = 0.97
Iteration 15: Success rate = 0.95
Iteration 16: Success rate = 0.95
Iteration 17: Success rate = 0.98
Iteration 18: Success rate = 0.95
Iteration 19: Success rate = 0.95
Iteration 20: Success rate = 0.97
Iteration 21: Success rate = 0.95
Iteration 22: Success rate = 0.98
Iteration 23: Success rate = 0.97
Iteration 24: Success rate = 0.92
Iteration 25: Success rate = 0.98
Iteration 26: Success rate = 0.97
Iteration 27: Success rate = 0.98
Iteration 28: Success rate = 0.98
Iteration 29: Success rate = 0.98
Iteration 30: Success rate = 1.00
Iteration 31: Success rate = 0.92
Iteration 32: Success rate = 0.98
Iteration 33: Success rate = 0.93
Iteration 34: Success rate = 0.98
Iteration 35: Success rate = 0.97
Iteration 36: Success rate = 0.95
Iteration 37: Success rate = 0.93
Iteration 38: Success rate = 0.98
Iteration 39: Success rate = 0.93
Iteration 40: Success rate = 0.92
Iteration 41: Success rate = 0.98
Iteration 42: Success rate = 1.00
Iteration 43: Success rate = 0.95
Iteration 44: Success rate = 0.95
Iteration 45: Success rate = 0.95
Iteration 46: Success rate = 0.98
Iteration 47: Success rate = 0.95
Iteration 48: Success rate = 0.98
Iteration 49: Success rate = 0.98
Iteration 50: Success rate = 0.92
Iteration 51: Success rate = 1.00
Iteration 52: Success rate = 0.95
Iteration 53: Success rate = 0.98
```





Project contributions

Prajwal Srinivas: Ps1458

- Bot 1 Algorithm Development
- Bot 2 Algorithm Development
- Grid Generation

Tejaswini Abburi: Ta633

- UCS Algorithm Development (Bot 3)
- A* Algorithm and GBfs Development-Hybrid Search (Bot 4)

Joint Contributions:

- Report Writing (Tejaswini Abburi and Prajwal Srinivas)
- Testing (Tejaswini Abburi and Prajwal Srinivas, split equally)
- Data Analysis: Collaboratively analysed simulation data to derive insights and conclusions regarding bot performance.(Prajwal Srinivas and Tejaswini Abburi)