جامعة جدة
University of Jeddah

Subject: Ai Fundamentals

Subject Code: CCAI-221

Exit Search Game

using A* star and Dijkstra’s Algorithm

Description of Game:

The ‘Exit Search Game’ originally is a game that involves 1 player (single agent). The goal is for the player to move the turtle to the red dot without touching any of the grey boxes (obstacles). If the turtle touches the obstacles, it returns to the starting point.

Now, the game includes 2 different turtles (multi agent) with each a different search algorithm to help it reach the red dot without interference of a player. It also uses a timer to count how many seconds each turtle takes to reach the red dot. At the end, a conclusion of the comparison of time is displayed stating which turtle reaches the red dot faster. The arrangement of obstacles changes every time the algorithm runs so each algorithm can display its results in terms of efficiency. The game is displayed with an interface.

Problems Solved using these Search Algorithms:

Techniques used in this game help in pathfinding and navigation. Some applications are used in robots that need to navigate in environments autonomously. GPS and mapping services like Google Maps also use these algorithms to help us find the shortest and fastest route to our destinations, taking into considerations things like traffic density and closed roads. The non-Player Characters (NPCs) in video games use them to navigate their way intelligently in the game world.

It is also used in internet routing and telecommunications to determine the best paths to data packets and best routes for signal transmission. The logistics field and disaster management and rescue operations all benefit from such algorithms.

Algorithms used:

- **A* Algorithm**: Green turtle, which moves first, uses this algorithm to reach the red dot. The A* algorithm is a heuristic-based search algorithm that finds the shortest path from a start node to a goal node. It is represented as $f(n) = g(n) + h(n)$ where $g(n)$ is the estimated cost to reach the goal node and $h(n)$ is the heuristic distance. This algorithm is more efficient most of the time as you will see; green turtle wins more times.
- **Dijkstra’s Algorithm**: Blue turtle, which moves next, uses this algorithm to reach the red dot. The Dijkstra’s algorithm is a search algorithm used in weighted graphs that finds the shortest path from start node to goal node. It starts from root node and starts exploring nodes that have the smallest distance, but still checks routes not taken in case they lead to the shortest path. This algorithm is efficient to find shortest path without preference unlike the heuristic approach but not efficient timewise.

Description of Code:

Initialization:

The code starts with initializing the screen object which is the interface of the game. The screen contains the title “Exit Search Game”, sets the background color, size of screen, and a tracer for the animation control.

Timer display is initialized next. There is a separate timer for each turtle coupled with some functions to assist in the game. The functions include hiding the turtle, lifting pen up so turtle does not draw while moving, going to coordinates, and writing the turtle’s position with given alignment and font.

The green turtle is created and given a speed of 5, specified the shape to ‘turtle’ and color to green. Also, the penup() function is used to not draw while moving. A starting position is given by (-380,0).

The red dot (exit) is located (380,0) opposite the green turtle.

The obstacles are made next. A while loop is made to make obstacles. However, each obstacle is checked so no obstacles are near turtles or exit. Also, no obstacles are overlapping.

A* Algorithm:

The graph dictionary stores the cost to move to each position. If position is near an obstacle the cost is raised to infinity. Otherwise, it is set to 20.

A queue is initialized to hold the starting position and ending with distance 0. Another queue is initialized to store the shortest distance to each position from the start. While the queue is not empty, the position of the smallest distance is popped and returned. If the position the turtle reaches the red dot, the loop breaks. For each new position, neighboring positions are considered by calculating the distance and updating the queue. The distance needs to be shorter than the previously known distance and the heuristic will be added.

A function called a_star is called to get distances to each position from the start. The function helps construct a path from the goal to the start by selecting the neighbor with minimum distance.

Dijkstra’s Algorithm:

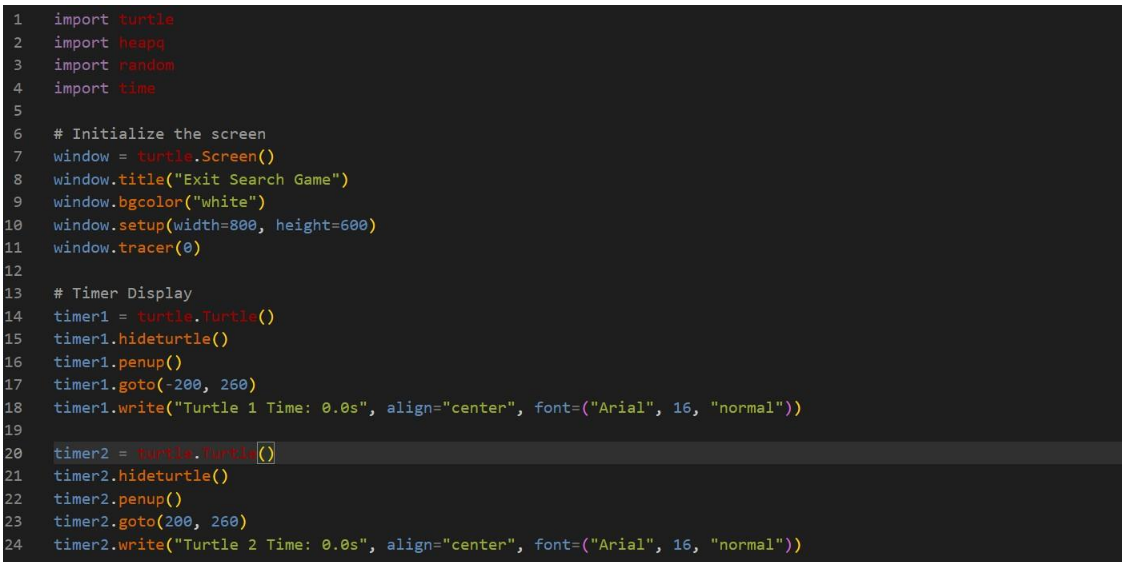
Just like the A* algorithm, the graph dictionary stores the cost to move to each position. If position is near an obstacle the cost is raised to infinity. Otherwise, it is set to 20.

This algorithm also uses the same priority queues. A queue is initialized to hold the starting position and ending with distance 0. Another queue is initialized to store the shortest distance to each position from the start. The distance needs to be shorter than the previously known distance and no heuristic will be added. Dijkstra’s search only finds the closest node and expands it. That’s how it finds the shortest path.

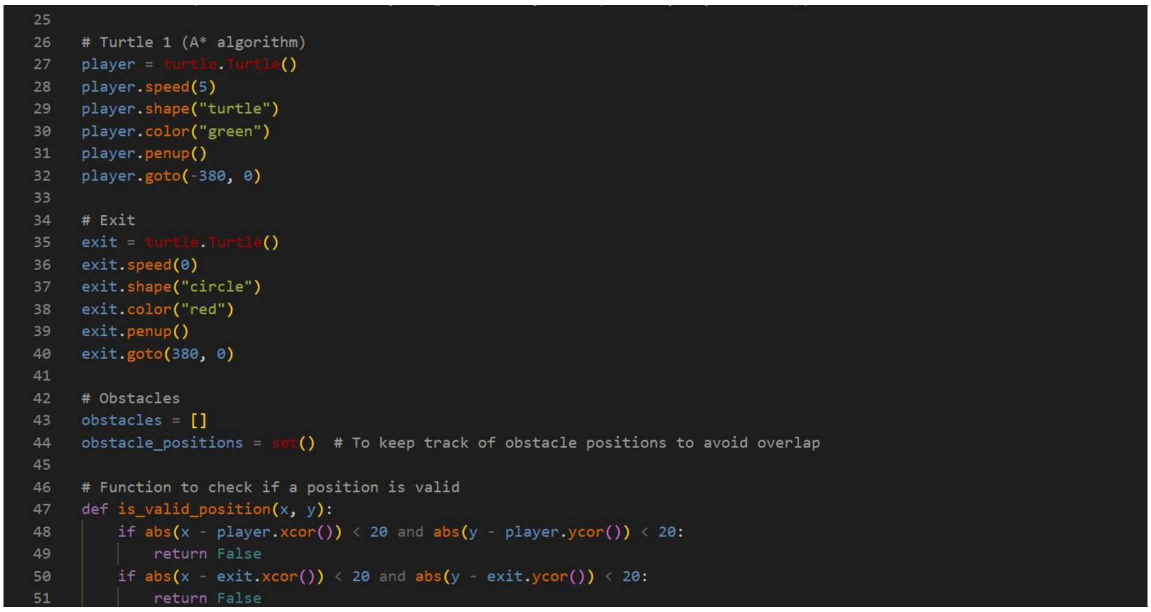
The Dijkstra function is called. It gets the distances to each position from the start. The path is constructed, and blue turtle moves along it.

Main:

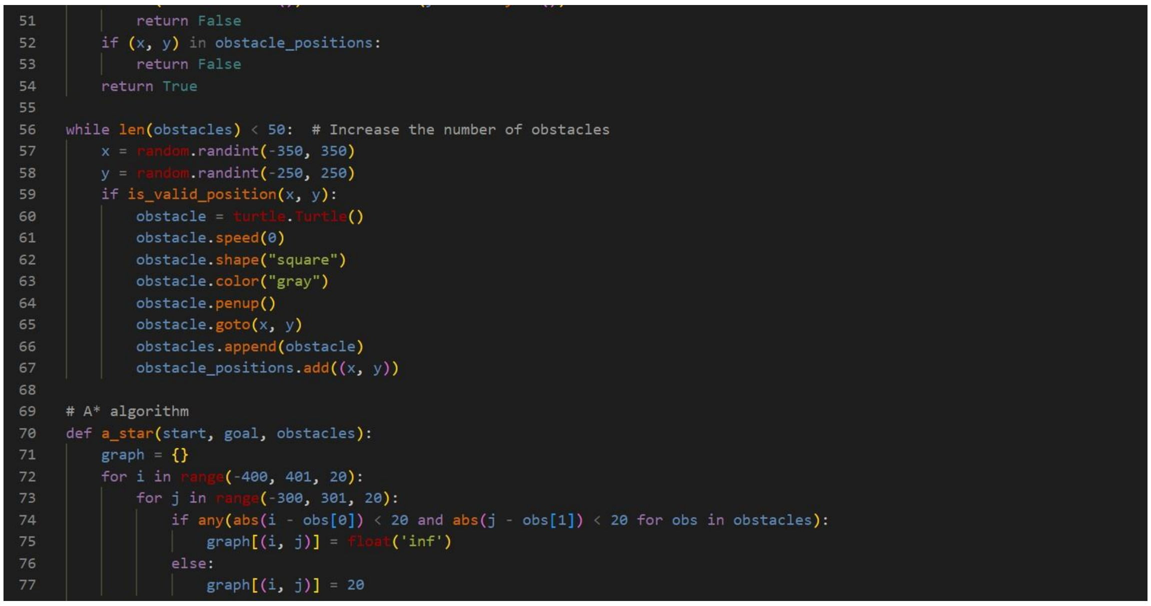
The timing of each turtle is recorded while each turtle gets a turn in finding the shortest path and moving towards it. It displays time using timer1 and timer2 turtles. A comparison is done using the 2 turtle times to conclude who is faster. The result is displayed in the center of the screen. A function is used to keep the window open until the user closes it manually.



```python
import turtle
import heapq
import random
import time

# Initialize the screen
window = turtle.Screen()
window.title("Exit Search Game")
window.bgcolor("white")
window.setup(width=800, height=600)
window.tracer(0)

# Timer Display
timer1 = turtle.Turtle()
timer1.hideturtle()
timer1.penup()
timer1.goto(-200, 260)
timer1.write("Turtle 1 Time: 0.0s", align="center", font=("Arial", 16, "normal"))

timer2 = turtle.Turtle()
timer2.hideturtle()
timer2.penup()
timer2.goto(200, 260)
timer2.write("Turtle 2 Time: 0.0s", align="center", font=("Arial", 16, "normal"))
```

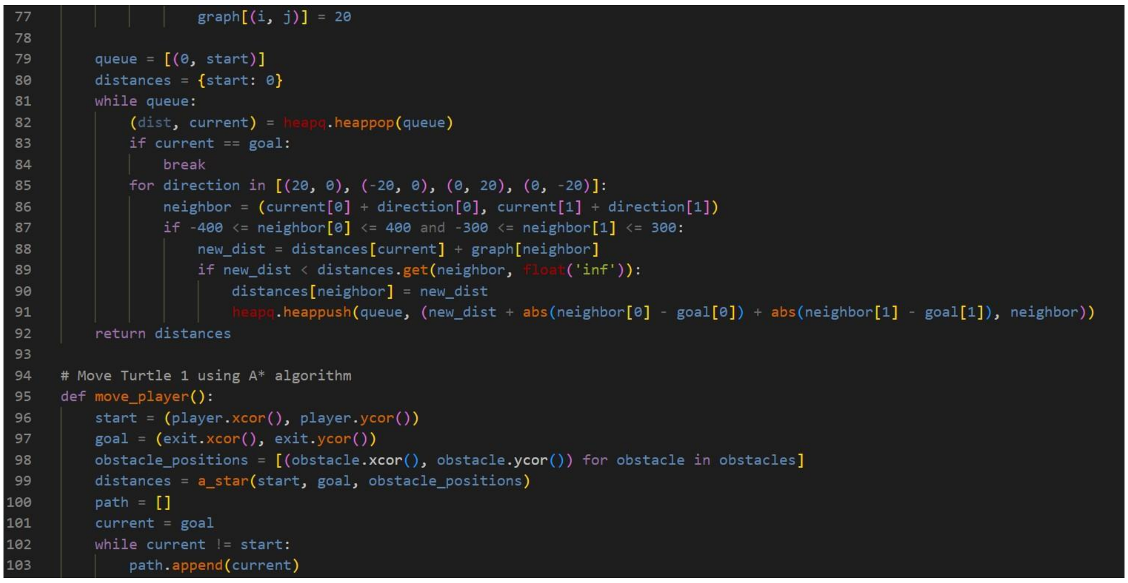


```python

# Turtle 1 (A* algorithm)
player = turtle.Turtle()
player.speed(5)
player.shape("turtle")
player.color("green")
player.penup()
player.goto(-380, 0)

# Exit
exit = turtle.Turtle()
exit.speed(0)
exit.shape("circle")
exit.color("red")
exit.penup()
exit.goto(380, 0)

# Obstacles
obstacles = []
obstacle_positions = set()  # To keep track of obstacle positions to avoid overlap

# Function to check if a position is valid
def is_valid_position(x, y):
    if abs(x - player.xcor()) < 20 and abs(y - player.ycor()) < 20:
        return False
    if abs(x - exit.xcor()) < 20 and abs(y - exit.ycor()) < 20:
        return False
```



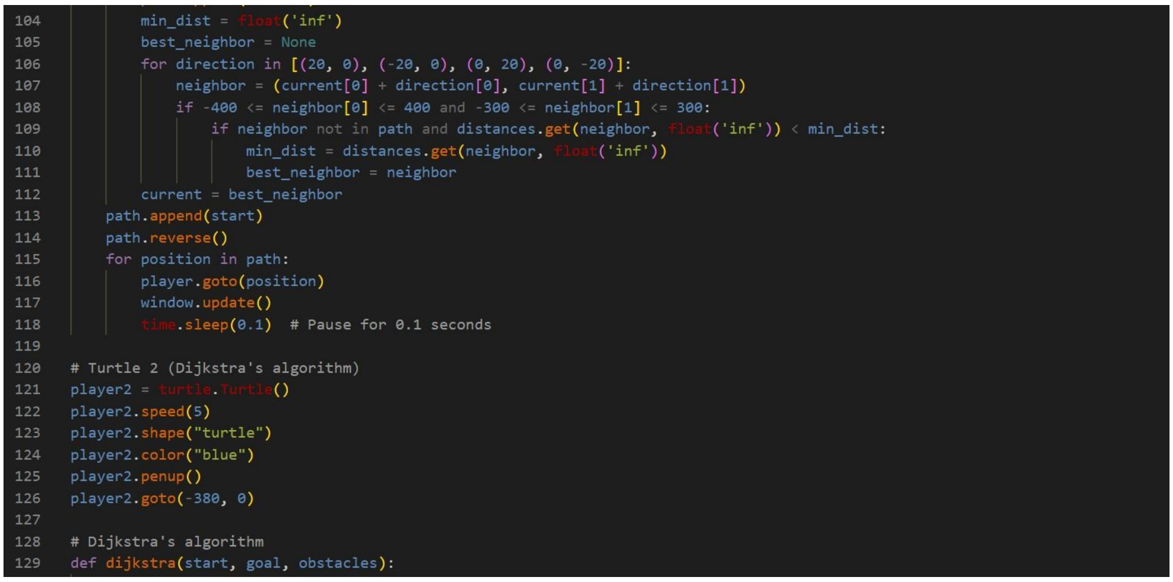
```python
        return False
```

```python
    if (x, y) in obstacle_positions:
        return False
    return True

while len(obstacles) < 50:  # Increase the number of obstacles
    x = random.randint(-350, 350)
    y = random.randint(-250, 250)
    if is_valid_position(x, y):
        obstacle = turtle.Turtle()
        obstacle.speed(0)
        obstacle.shape("square")
        obstacle.color("gray")
        obstacle.penup()
        obstacle.goto(x, y)
        obstacles.append(obstacle)
        obstacle_positions.add((x, y))

# A* algorithm
def a_star(start, goal, obstacles):
    graph = {}
    for i in range(-400, 401, 20):
        for j in range(-300, 301, 20):
            if any(abs(i - obs[0]) < 20 and abs(j - obs[1]) < 20 for obs in obstacles):
                graph[(i, j)] = float('inf')
            else:
                graph[(i, j)] = 20
```

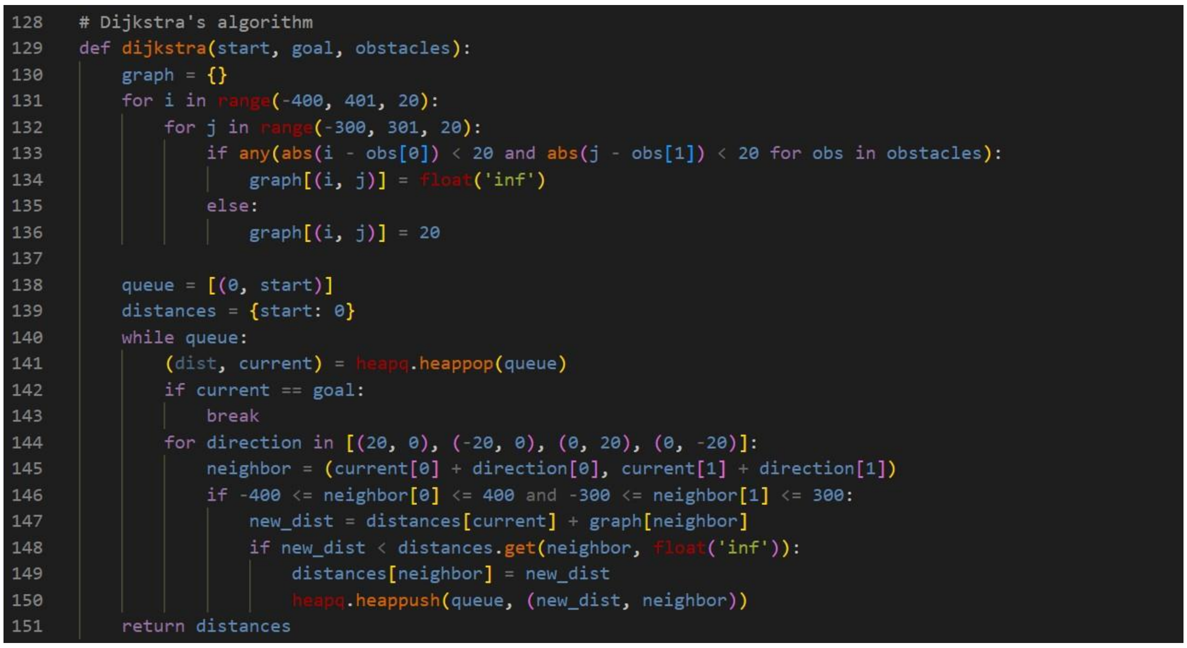


```python
                graph[(i, j)] = 20
```

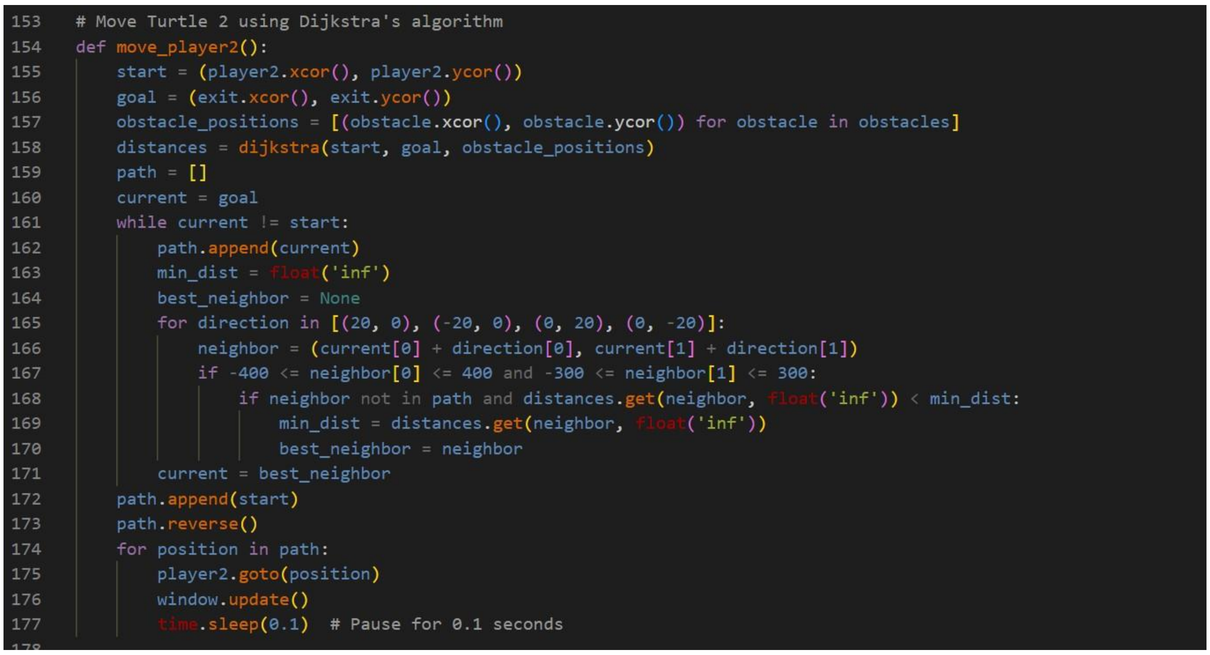
```python

    queue = [(0, start)]
    distances = {start: 0}
    while queue:
        (dist, current) = heapq.heappop(queue)
        if current == goal:
            break
        for direction in [(20, 0), (-20, 0), (0, 20), (0, -20)]:
            neighbor = (current[0] + direction[0], current[1] + direction[1])
            if -400 <= neighbor[0] <= 400 and -300 <= neighbor[1] <= 300:
                new_dist = distances[current] + graph[neighbor]
                if new_dist < distances.get(neighbor, float('inf')):
                    distances[neighbor] = new_dist
                    heapq.heappush(queue, (new_dist + abs(neighbor[0] - goal[0]) + abs(neighbor[1] - goal[1]), neighbor))
    return distances

# Move Turtle 1 using A* algorithm
def move_player():
    start = (player.xcor(), player.ycor())
    goal = (exit.xcor(), exit.ycor())
    obstacle_positions = [(obstacle.xcor(), obstacle.ycor()) for obstacle in obstacles]
    distances = a_star(start, goal, obstacle_positions)
    path = []
    current = goal
    while current != start:
        path.append(current)
```



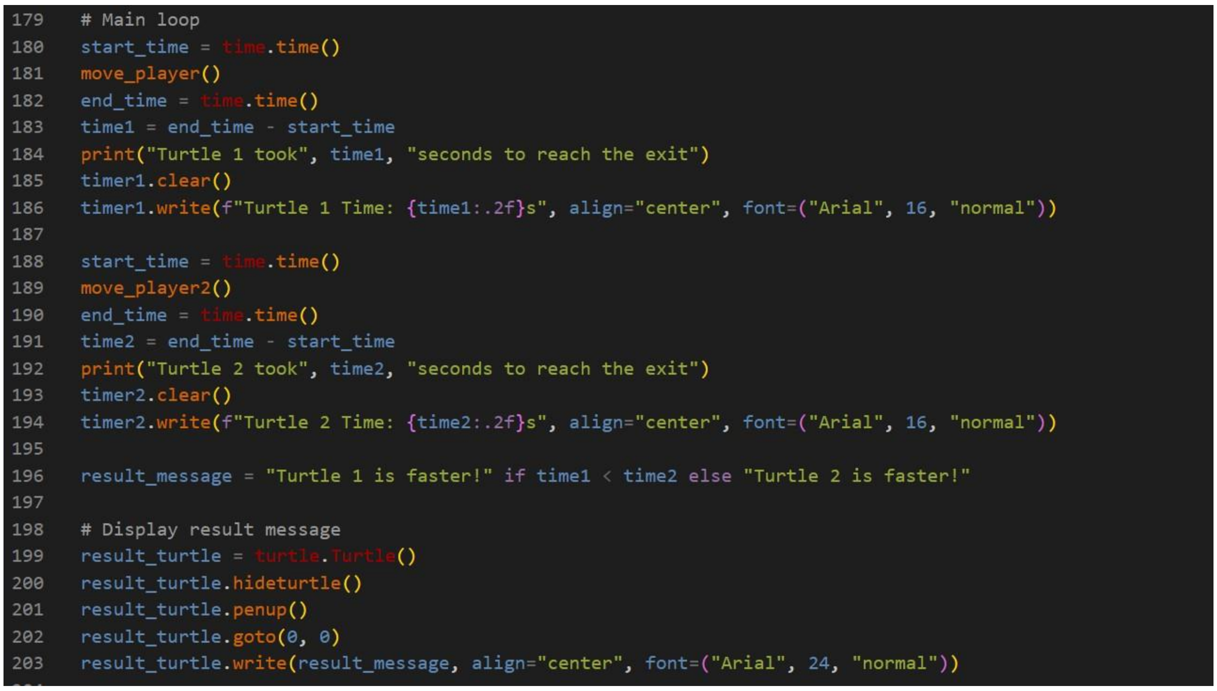
```python
        min_dist = float('inf')
        best_neighbor = None
        for direction in [(20, 0), (-20, 0), (0, 20), (0, -20)]:
            neighbor = (current[0] + direction[0], current[1] + direction[1])
            if -400 <= neighbor[0] <= 400 and -300 <= neighbor[1] <= 300:
                if neighbor not in path and distances.get(neighbor, float('inf')) < min_dist:
                    min_dist = distances.get(neighbor, float('inf'))
                    best_neighbor = neighbor
        current = best_neighbor
    path.append(start)
    path.reverse()
    for position in path:
        player.goto(position)
        window.update()
        time.sleep(0.1)  # Pause for 0.1 seconds

# Turtle 2 (Dijkstra's algorithm)
player2 = turtle.Turtle()
player2.speed(5)
player2.shape("turtle")
player2.color("blue")
player2.penup()
player2.goto(-380, 0)

# Dijkstra's algorithm
def dijkstra(start, goal, obstacles):
```



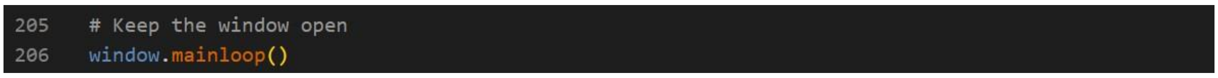
```python
# Dijkstra's algorithm
def dijkstra(start, goal, obstacles):
    graph = {}
    for i in range(-400, 401, 20):
        for j in range(-300, 301, 20):
            if any(abs(i - obs[0]) < 20 and abs(j - obs[1]) < 20 for obs in obstacles):
                graph[(i, j)] = float('inf')
            else:
                graph[(i, j)] = 20

    queue = [(0, start)]
    distances = {start: 0}
    while queue:
        (dist, current) = heapq.heappop(queue)
        if current == goal:
            break
        for direction in [(20, 0), (-20, 0), (0, 20), (0, -20)]:
            neighbor = (current[0] + direction[0], current[1] + direction[1])
            if -400 <= neighbor[0] <= 400 and -300 <= neighbor[1] <= 300:
                new_dist = distances[current] + graph[neighbor]
                if new_dist < distances.get(neighbor, float('inf')):
                    distances[neighbor] = new_dist
                    heapq.heappush(queue, (new_dist, neighbor))
    return distances
```



```python
# Move Turtle 2 using Dijkstra's algorithm
def move_player2():
    start = (player2.xcor(), player2.ycor())
    goal = (exit.xcor(), exit.ycor())
    obstacle_positions = [(obstacle.xcor(), obstacle.ycor()) for obstacle in obstacles]
    distances = dijkstra(start, goal, obstacle_positions)
    path = []
    current = goal
    while current != start:
        path.append(current)
        min_dist = float('inf')
        best_neighbor = None
        for direction in [(20, 0), (-20, 0), (0, 20), (0, -20)]:
            neighbor = (current[0] + direction[0], current[1] + direction[1])
            if -400 <= neighbor[0] <= 400 and -300 <= neighbor[1] <= 300:
                if neighbor not in path and distances.get(neighbor, float('inf')) < min_dist:
                    min_dist = distances.get(neighbor, float('inf'))
                    best_neighbor = neighbor
        current = best_neighbor
    path.append(start)
    path.reverse()
    for position in path:
        player2.goto(position)
        window.update()
        time.sleep(0.1)  # Pause for 0.1 seconds
```

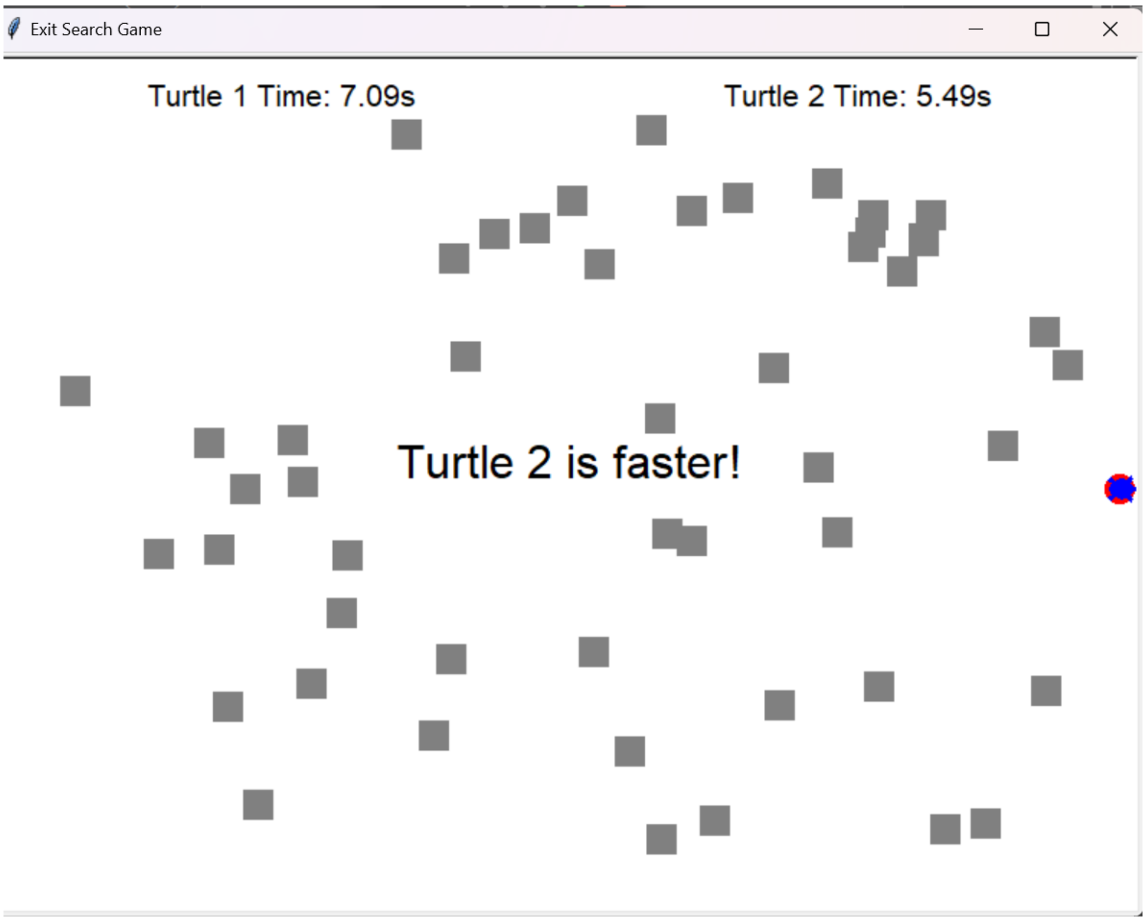


```python
# Main loop
start_time = time.time()
move_player()
end_time = time.time()
time1 = end_time - start_time
print("Turtle 1 took", time1, "seconds to reach the exit")
timer1.clear()
timer1.write(f"Turtle 1 Time: {time1:.2f}s", align="center", font=("Arial", 16, "normal"))

start_time = time.time()
move_player2()
end_time = time.time()
time2 = end_time - start_time
print("Turtle 2 took", time2, "seconds to reach the exit")
timer2.clear()
timer2.write(f"Turtle 2 Time: {time2:.2f}s", align="center", font=("Arial", 16, "normal"))

result_message = "Turtle 1 is faster!" if time1 < time2 else "Turtle 2 is faster!"

# Display result message
result_turtle = turtle.Turtle()
result_turtle.hideturtle()
result_turtle.penup()
result_turtle.goto(0, 0)
result_turtle.write(result_message, align="center", font=("Arial", 24, "normal"))
```



```python
# Keep the window open
window.mainloop()
```

Result 1:

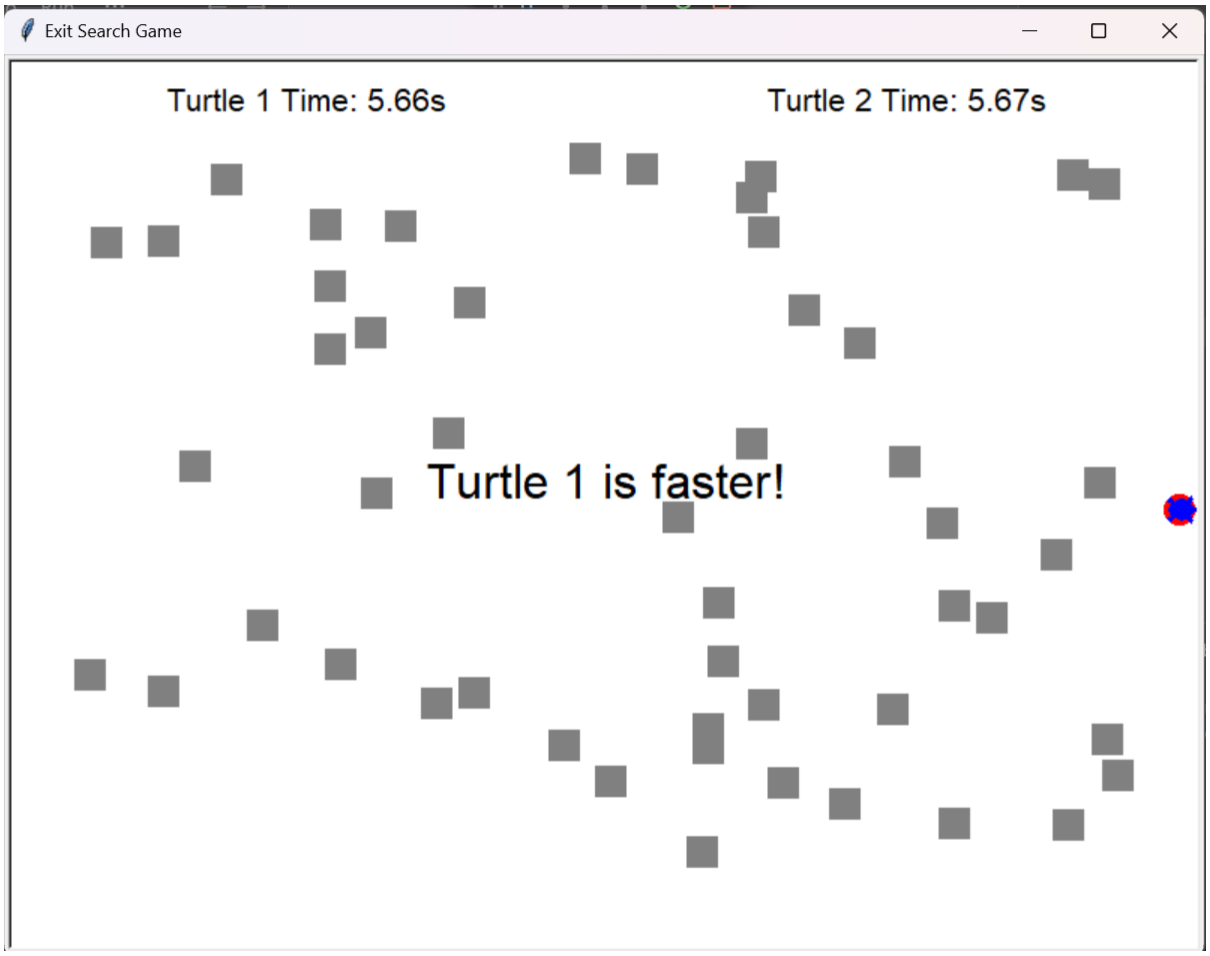


Turtle 1 Time: 7.09s

Turtle 2 Time: 5.49s

Turtle 2 is faster!

Result 2:



Turtle 1 Time: 5.66s

Turtle 2 Time: 5.67s

Turtle 1 is faster!