A* (A-Star) 路徑規劃演算法：

新增 euclidean_distance 方法

1. 計算從某個節點到終點的歐幾里得距離 (Euclidean Distance)，作為啟發函數 h(n)，取代原有程式直接使用 utils.distance(start, goal)。

```
def euclidean_distance(self, node, goal):
    return np.sqrt((goal[0] - node[0])**2 + (goal[1] - node[1])**2)
```

2. TODO內容：

新增 A* 搜索過程，包括：

- 選擇 f(n) = g(n) + h(n) 最小代價的節點

```
while self.queue:
    # Select the node with the minimum f(n) = g(n) + h(n)
    current = min(self.queue, key=lambda n: self.g[n] + self.h[n])
    self.queue.remove(current)

    # If the goal is reached
    if current == goal:
        self.goal_node = current
        break
```

- 擴展尋找周圍節點

```
neighbors = [
    (current[0] + inter, current[1]),  # Right
    (current[0] - inter, current[1]),  # Left
    (current[0], current[1] + inter),  # Down
    (current[0], current[1] - inter),  # Up
    ]
```

- 確保鄰近節點不超出地圖範圍

```
if not (0 <= neighbor[0] < self.map.shape[1] and 0 <= neighbor[1] < self.map.shape
    continue
```

- 確保節點不在障礙物內，進行碰撞檢測 (確保不碰撞障礙物)

```
if self.map[neighbor[1], neighbor[0]] < 0.2:  # Adjusted threshold
```

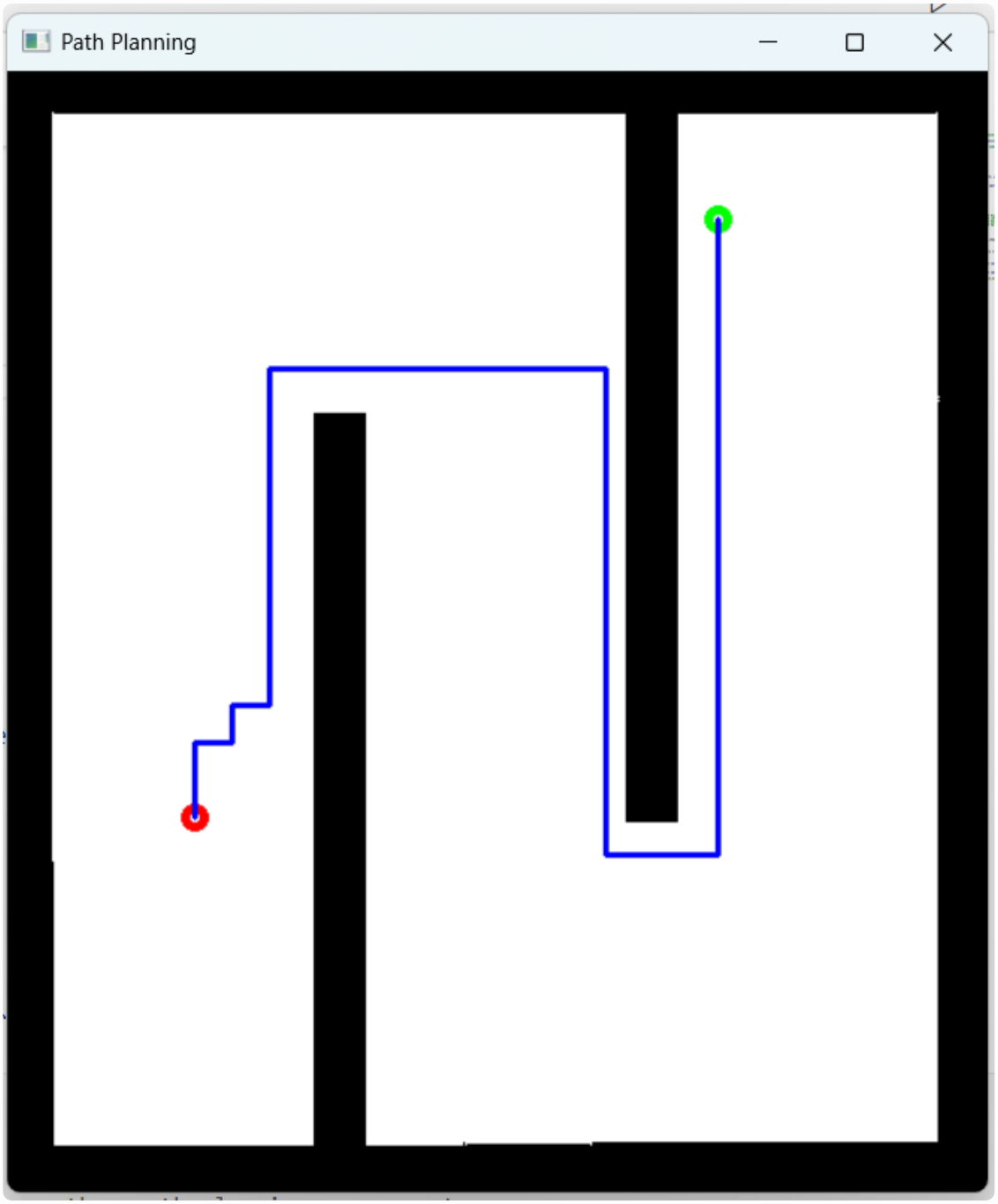
```
    continue
```

- 更新 g(n) (成本) 並記錄父節點
    - 計算從 current 到 neighbor 的移動成本 g(n)：

```
move_cost = 1.414 if (neighbor[0] != current[0] and neighbor[1] != current[1]) els
new_g = self.g[current] + move_cost
```

- 若 neighbor 是新的節點或發現更短路徑，則更新 g(n), h(n), parent，並加入 queue：

```
if neighbor not in self.g or new_g < self.g[neighbor]:
    self.g[neighbor] = new_g
    self.h[neighbor] = self.euclidean_distance(neighbor, goal)  # Use Euclidean he
    self.parent[neighbor] = current
    self.queue.append(neighbor)
```



RRT* (RRT-Star) 路徑規劃演算法：

1. 增加終點偏向機率 (Goal Bias)

新增

```
self.goal_bias = 0.3  # 初始目標偏向機率
```

變更 _random_node 方法，改為動態調整機率，隨著迭代次數增加，提高選擇目標點的機率，最大可達 90%，來加速收斂：

```
def _random_node(self, goal, shape, iteration):
    if np.random.rand() < min(0.9, self.goal_bias + iteration / 10000):
        return (float(goal[0]), float(goal[1]))
    else:
        rx = float(np.random.randint(int(shape[1])))
        ry = float(np.random.randint(int(shape[0])))
        return (rx, ry)
```

2. TODO： Re-Parent & Re-Wire

新增鄰近半徑變數：

```
radius = extend_len * 2.5  # 定義鄰近半徑
```

Re-Parent & Re-Wire 的程式邏輯：

```
best_parent = near_node
min_cost = self.cost[near_node] + cost
neighbors = [n for n in self.ntree if utils.distance(n, new_node) < radius]

# 嘗試找到最佳父節點
for n in neighbors:
    if n in self.cost:
        temp_cost = self.cost[n] + utils.distance(n, new_node)
        if temp_cost < min_cost and not self._check_collision(n, new_node):
            best_parent = n
            min_cost = temp_cost

self.ntree[new_node] = best_parent
self.cost[new_node] = min_cost

# 重新調整其他鄰近節點
for n in neighbors:
    if n in self.cost:
        new_cost = self.cost[new_node] + utils.distance(new_node, n)
        if new_cost < self.cost[n] and not self._check_collision(new_node, n):
            self.ntree[n] = new_node
            self.cost[n] = new_cost
```

3. 增加路徑平滑化：

新增：

```
self.smoothing_iterations = 15  # 路徑平滑次數
```

新增 smooth_path 方法：

```
def smooth_path(self, path):
    for _ in range(self.smoothing_iterations):
        if len(path) <= 2:
            break
        i, j = sorted(np.random.randint(0, len(path), size=2))
        if j - i > 1 and not self._check_collision(path[i], path[j]):
            path = path[:i + 1] + path[j:]
    return path
```

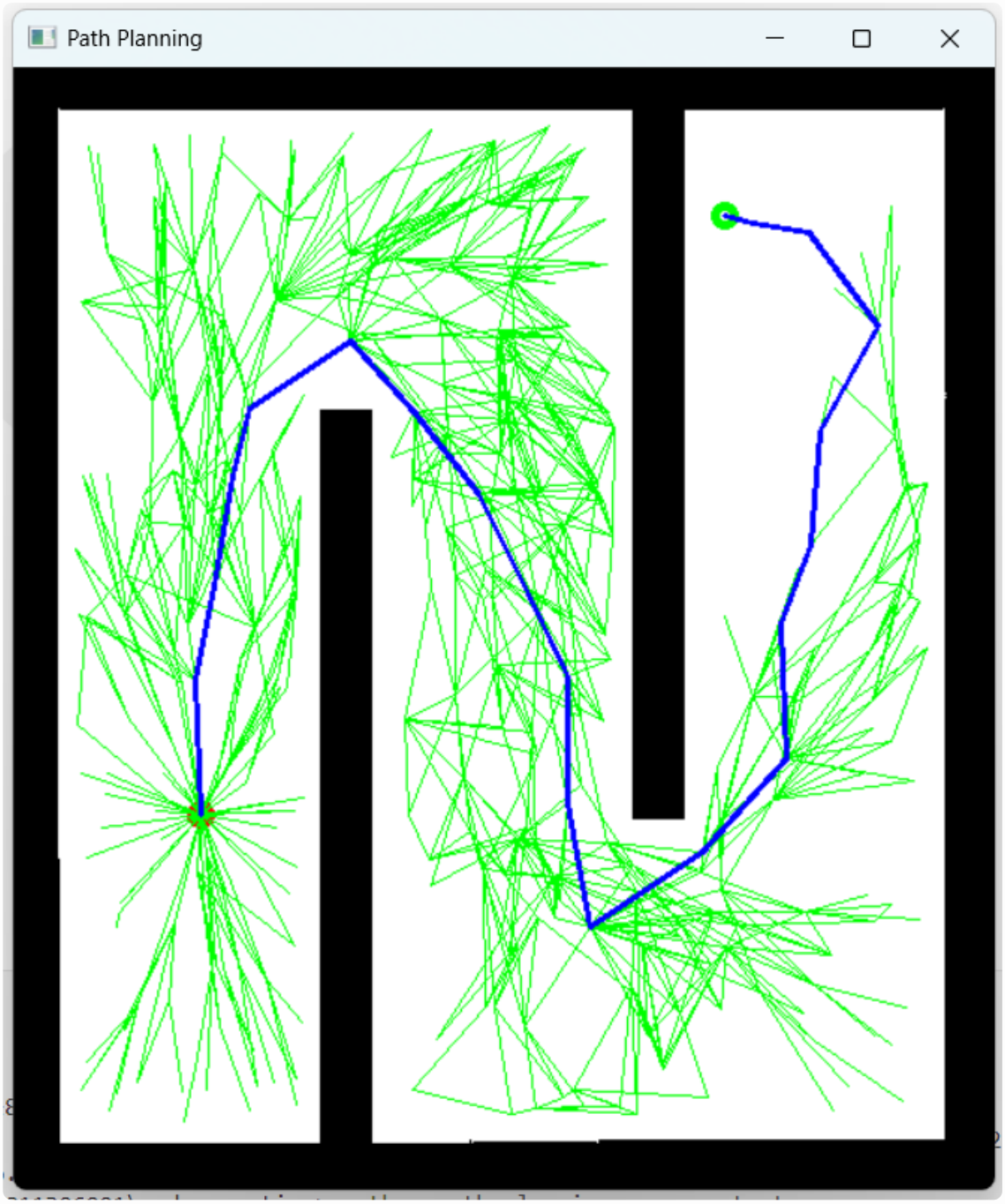
4. 修改提取路徑方法

```
def extract_path(self, goal_node, goal):
    path = []
    n = goal_node
    while n:
        path.insert(0, n)
        n = self.ntree[n]
    path.append(goal)
    return path
```

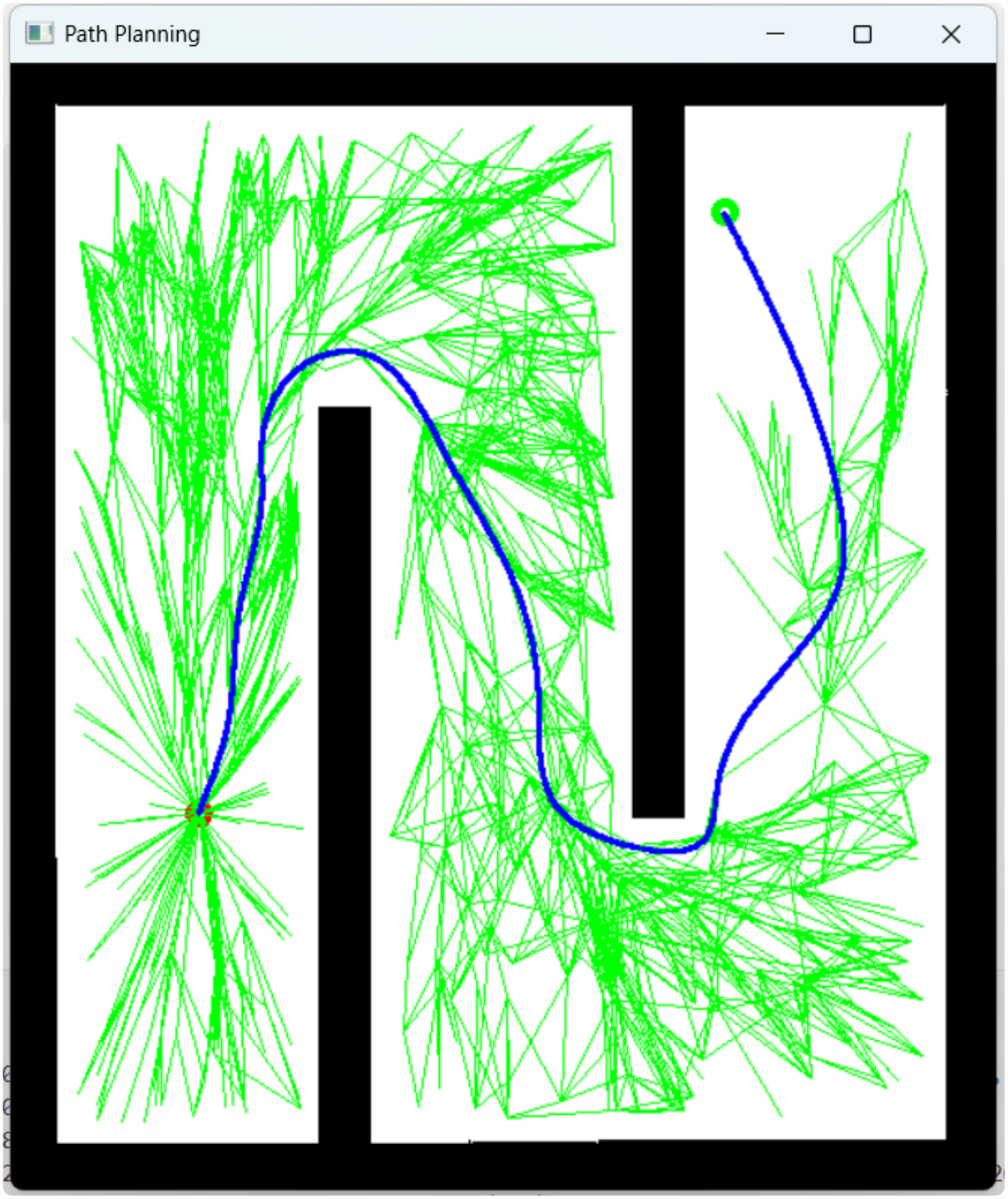
增加

```
path = self.extract_path(goal_node, goal)
path = self.smooth_path(path)
return path
```

python path_planning.py -p rrt_star



python path_planning.py -p rrt_star –smooth


Path Planning