Path Planning Benchmark: Grid-Based Algorithms in Python

Made by: Anshuman Venkata Raghavan, K Tharachand Chowdhary

This documentation presents a comprehensive benchmark and simulator for visualizing pathfinding algorithms using a grid-based environment implemented in Python. The project is designed for learners, educators, and robotics enthusiasts to better understand and compare algorithmic strategies in navigation and obstacle avoidance. A custom "Potential*" Algorithm has also been implemented here, which can be benchmarked with other standard algorithms.

The system supports:

- Visual, real-time simulation via PyGame.
- Interactive controls to design custom environments.
- Detailed algorithm performance tracking (runtime, path length, node exploration).
- Three algorithm modes: Breadth-First Search (BFS), A* Search, and a Potential* algorithm (which uses potential field search, with smart A* fallback.)

  Whether you're exploring AI, robotics, or game development, this tool provides a powerful way to study how different algorithms behave in the same scenario.

Path Planning Fundamentals

Definition
Path planning computes a collision-free route from a start to a goal location within an environment containing obstacles. It aims to balance:

- Optimality: Find the shortest or most efficient path.
- Efficiency: Minimize computation and memory.
- Safety: Avoid collisions with obstacles.

Applications
Path planning is critical across various domains:

- Robotics: Factory automation, warehouse logistics.
- Autonomous Vehicles: Navigation through urban environments.
- Game AI: Controlling NPC behavior.
- Drones: Surveillance and delivery.

Challenges

- Local Minima: Some algorithms (like potential fields) can get stuck near obstacles.
- High Dimensionality: Increases the complexity of the problem.
- Dynamic Environments: Require real-time decision-making.

Grid-based path planning simplifies the environment into discrete 2D cells. This allows for easier implementation of search algorithms and visual debugging.

PyGame Environment Implementation

Grid Setup

```
GRID_SIZE = 30
CELL_SIZE = 20

grid = np.zeros((GRID_SIZE, GRID_SIZE), dtype=int)
# 0=Empty, 1=Start, 2=Goal, 3=Wall, 4=Visited, 5=Path, 6=Fallback
```

- Grid is implemented as a 2D numpy array.
- Each integer represents a state (empty, wall, visited, etc).

Grid Visualization

```
def draw_grid():
    for y in range(GRID_SIZE):
        for x in range(GRID_SIZE):
            color = COLOR_MAP[grid[y, x]]
            pygame.draw.rect(screen, color, (x * CELL_SIZE, y * CELL_SIZE,
CELL_SIZE, CELL_SIZE))
```

Draws the entire environment, updating cell colors based on state values.

Breadth-First Search (BFS)

Breadth-First Search explores nodes level by level, ensuring the shortest path in unweighted grids. It uses a queue to track the next nodes to visit.

Core Loop

```
def bfs(start, goal):
    queue = deque([start])
    visited = set([start])
    came_from = {}
```

- `queue`: Stores frontier nodes.
- `visited`: Prevents revisiting nodes.
- `came_from`: Tracks node parents for path reconstruction.

```
    while queue:
```

```
        current = queue.popleft()

        if current == goal:
            return reconstruct_path(came_from, current)
```

- Terminates early if goal is reached.

```
        for neighbor in get_neighbors(current):
            if neighbor not in visited:
                visited.add(neighbor)
                came_from[neighbor] = current
                queue.append(neighbor)

    return None  # No path found
```

- Explores all valid neighbors.

A* Search

A* enhances BFS using a heuristic function to prioritize nodes likely to reach the goal faster.

Initialization

```
def a_star(start, goal):
    open_set = PriorityQueue()
    open_set.put((0, start))

    came_from = {}
    g_score = {start: 0}
    f_score = {start: heuristic(start, goal)}
```

- `open_set`: Priority queue with (f_score, node).
- `g_score`: Tracks path cost from start.
- `f_score`: Estimated total cost.

Main Loop

```
    while not open_set.empty():
        current = open_set.get()[1]

        if current == goal:
            return reconstruct_path(came_from, current)
```

- Extracts node with lowest estimated cost.
- Stops when goal is found.

```
        for neighbor in get_neighbors(current):
            tentative_g = g_score[current] + 1
```

```
            if neighbor not in g_score or tentative_g < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g
                f_score[neighbor] = tentative_g + heuristic(neighbor, goal)
                open_set.put((f_score[neighbor], neighbor))

    return None
```

- The algorithm constantly checks the f_score of the current cells neighbours and travels to the cell with the lowest score.
- The Manhattan heuristics allow us to obtain optimal paths in an environment where no diagonal moves are allowed.

Hybrid: Potential Field + A* Recovery (Potential*)

This hybrid approach uses artificial potential fields for fast, local motion and invokes A* only when the agent is stuck. Once A* navigates out of the stuck region, the agent switches back to Potential field search, for faster navigation.

Force-Based Navigation

```
def compute_force(current, goal, obstacles):
    F_att = -k_att * (current - goal)
    F_rep = np.zeros_like(F_att)

    for obs in obstacles:
        diff = current - obs
        dist = np.linalg.norm(diff)
        if dist < d_safe:
            F_rep += k_rep * (1/dist - 1/d_safe) * (diff / (dist ** 3))

    return F_att + F_rep
```

- Attraction pulls the agent toward the goal.
- Repulsion pushes it away from nearby obstacles.

Detecting Stagnation

```
def is_stuck(position_history):
    if len(position_history) < HISTORY_LENGTH:
        return False
    recent_movement = np.linalg.norm(position_history[-1] -
position_history[0])
    return recent_movement < STAGNATION_THRESHOLD
```

- Detects lack of movement over recent steps.
- Prevents false positives by using a history buffer.

Fallback via A*

```
def fallback_a_star(current, goal):
    return a_star(current, goal)
```

- A* is triggered from a few steps before the stuck point.
- The fallback path is stitched to the existing trajectory.

Tuning Parameters

```
k_att = 1.0
k_rep = 5.0
d_safe = 3.0
STAGNATION_THRESHOLD = 2.0
HISTORY_LENGTH = 10
```

- Control the agent's sensitivity to goals and obstacles.

Execution & UI

- Press Left Click to place Start, Goal and Obstacles.
- ▶ Run the selected algorithm.
- Compare timings of multiple runs.
- R- Reset the grid.

Performance Benchmarks

Algorithm	Nodes Explored	Path Length	Time (ms)	Success Rate
A*	1,200 ± 300	Optimal	45 ± 10	100%
BFS	3,500 ± 500	Optimal	120 ± 30	100%
Potential*	400 ± 150	~105% Opt	35 ± 15	98%

Conclusion

This benchmark showcases the value of combining reactive navigation with heuristic search. The potential field handles fast local navigation, while A* guarantees escape from traps. PyGame enables clear visualization and extensibility. The custom Potential* Algorithm navigates open areas significantly faster than standard A*. Moreover in highly cluttered environments, it's performance is similar to standard A* algorithm. Overall this algorithm performs better in most use cases than A*.

Potential Enhancements

- Support for dynamic obstacles.
- Multi-agent planning.
- Terrain-aware cost maps.
- ML-based tuning for force parameters.
