ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/25sep776

A Comparative Study of Pathfinding Algorithms for Low-Cost Mobile Robots in Dynamic Environments

Shrikar Nagarajan[1]; Aarav Upreti[2]; Nathanael Dhanapal[3]; Nathan Dsouza[4]

[1]Delhi Private School Dubai
[2]Delhi Private School Dubai
[3]Delhi Private School Dubai
[4]Delhi Private School Dubai

Publication Date: 2025/09/27

Abstract: Pathfinding is critical in mobile robotics for enabling autonomous navigation from a start to a goal location while avoiding obstacles. This study implements six representative pathfinding algorithms – Dijkstra, A*, Breadth-First Search (BFS), Greedy Best-First Search, Bug1, and Bug2 – and compares their performance on grid-based maps under low-cost robot constraints (limited battery and sensing) and dynamic changes (moving obstacles). We simulate a two-dimensional grid world with static and dynamic obstacles, modeling a simple wheeled robot with limited sensors and a finite battery. Each algorithm is evaluated on key metrics: path length, computation time, battery usage (proportional to distance traveled and actions taken), success rate (reaching the goal without failure), and adaptability to environmental changes. Our results show that A* consistently yields the shortest path and fastest search time in static, known environments, while BFS and Dijkstra also find optimal paths, albeit with higher computational costs. Greedy Best-First Search often finds a path quickly but can produce suboptimal or invalid paths under complex scenarios. The simple Bug algorithms (Bug1 and Bug2) are robust to unknown obstacles (requiring only local sensing) and guarantee finding a path if one exists, albeit at the expense of significantly longer detours and greater energy consumption. In dynamic scenarios (moving obstacles), global planners (A*, Dijkstra) must replan or may fail, whereas reactive Bug planners naturally cope by following obstacle boundaries. Overall, A* performs best in static settings with sufficient compute, while simpler methods or hybrid strategies may be preferable for very low-cost robots or highly dynamic settings. Our comprehensive comparison highlights the trade-offs of each algorithm and guides the choice of planning strategy based on environmental demands and resource constraints.

Keywords: *Pathfinding Algorithms, Low-Cost Mobile Robots, Dynamic Environments, Dijkstra, Bug Algorithms, Resource-Constrained Robotics, Grid-Based Simulation, Battery-Efficient Planning.*

How to Cite: Shrikar Nagarajan; Aarav Upreti; Nathanael Dhanapal; Nathan Dsouza (2025) A Comparative Study of Pathfinding Algorithms for Low-Cost Mobile Robots in Dynamic Environments. *International Journal of Innovative Science and Research Technology*, 10(9), 1835-1841. https://doi.org/10.38124/ijisrt/25sep776

I. INTRODUCTION

Path planning is a fundamental capability in mobile robotics: a robot must autonomously move from its current position to a desired goal while avoiding obstacles. In many applications (industrial robots, warehouses, service robots, exploration), finding a feasible collision-free trajectory is critical for efficiency and safety. We define *low-cost* mobile robots as those with constrained resources: limited onboard computation, minimal sensors, and modest battery capacity. These constraints arise in educational robots, hobby platforms, or budget service robots. Dynamic environments refer to settings where obstacles and goals may change over time (e.g., moving objects, shifting layouts), requiring real-time replanning or reactive behaviors. Under these conditions, it is important to compare different planning algorithms to understand their suitability. Some algorithms assume full static maps and heavy computation, while others use only local sensing but might be inefficient or incomplete.

- *Global Planners*

Classical global planners like BFS, Dijkstra's, and A* assume complete map knowledge. Breadth-first search (BFS) is a brute-force uninformed search that expands nodes level by level and guarantees the shortest path in an unweighted grid. Dijkstra's algorithm generalizes this to weighted graphs, finding the minimum-cost paths from a start to all nodes. A* is a best-first search that uses a heuristic to prioritize nodes closer to the goal, combining the advantages of BFS and depth-first search; it finds optimal paths (with an admissible heuristic) more efficiently than Dijkstra. Greedy Best-First Search is similar to A* but uses only the heuristic (estimated

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/25sep776

distance to goal) as its evaluation function; it can be faster but does not guarantee optimal or even valid paths.

In contrast, Bug1 and Bug2 are simple sensor-based reactive algorithms. They assume *only local knowledge* of obstacles and a known goal position. A Bug algorithm repeatedly moves toward the goal until it encounters an obstacle, then follows the obstacle boundary (wall-following) according to rules. Bug1 fully circumnavigates the obstacle to find the point closest to the goal before departing, whereas Bug2 follows the obstacle until re-encountering the straight-line from start to goal (the "m-line") at a point closer to the goal than the initial hit point. Under reasonable assumptions (point robot, closed obstacles), Bug1 and Bug2 are *complete*: they will reach the goal if it is reachable. However, their paths can be very long, and they may waste battery circling obstacles.

➢ *Research Objectives and Gaps*

Our work implements all six algorithms in a common simulation to evaluate them on equal footing. We consider a low-cost differential-drive robot with a finite battery and simple sensors in a 2D grid that may include moving obstacles. We are motivated by the need to understand *trade-offs*: for instance, A* finds optimal paths but requires enough processing power and may not react to changes, whereas a Bug algorithm is cheap and reactive but inefficient. Prior work often focuses on one method or domain: for example, Spektor et al. implemented Bug1 and Bug2 in ROS/Gazebo, Maneev and Syryamkin optimized A* for mobile devices, and comparative studies have evaluated A*, RRT, PRM, etc. However, there is limited work directly comparing classical and simple algorithms under dynamic conditions and resource constraints. We differ from existing studies by simulating all six algorithms on the same low-cost robot model, injecting dynamic obstacles, and measuring energy/battery usage along with path and time performance.

Previous research has extensively studied individual path planning algorithms. Maneev and Syryamkin (2019) analyze BFS, DFS, Dijkstra, Greedy, and A*, proposing optimizations for small mobile devices; they note that A* is widely used in mobile robot routing but can be too computationally expensive for limited platforms, especially in dynamic environments. Spektor *et al.* (2024) implemented Bug1 and Bug2 on a TurtleBot3 in simulation, highlighting the importance of safe local planners in unknown 3D environments. Several comparative studies exist (e.g., Al-Zubaidi *et al.*, 2023, compared A*, RRT, PRM for pick-and-place robot paths) but often focus on high-end robots or different domains. Notably, our work integrates considerations like battery usage and dynamic replanning, which are seldom addressed jointly. As Dudzik and Rapalski (2023) emphasize, the choice of planning algorithm can significantly affect energy consumption; we extend this by comparing energy use across varied methods. Thus, this study builds on classic algorithm theory but uniquely evaluates performance under low-cost and dynamic conditions.

The full implementation and datasets used are available at GitHub Repository: https://github.com/AaravUp/pathfinding-algorithms

II. METHODOLOGY

Each algorithm was evaluated in a 2D grid-based simulation environment designed to mimic dynamic indoor conditions. The total task time was recorded across multiple trials, capturing both planning and execution phases for global planners, and execution-only for reactive algorithms.

A. Environment

➢ *Layout*

We simulate a 2D grid world of size $N \times N$ (e.g., 10×10 cells) to model the robot's workspace. Each cell is either free or occupied by an obstacle. In static tests, obstacles are fixed; in dynamic tests, one or more obstacles move along predefined or random trajectories between timesteps. For example, a moving block might oscillate or wander, forcing the robot to adapt or replan. Grid cells correspond to nodes in a graph, with edges connecting adjacent (4-way or 8-way) neighbors. Obstacles are implemented by removing those nodes or marking them impassable. This discrete representation allows straightforward implementation of BFS, Dijkstra, A*, etc., as well as a simple model for reactive behaviors.

➢ *Reflecting Real-World Scenarios*

We incorporate environmental uncertainties to reflect real low-cost scenarios.

- Slippage: When the robot attempts to move to a cell, there is a small probability (0.05%) that it does not move (to mimic wheel slip).
- Sensor Noise: Obstacle detection is not perfect; the robot might not see an obstacle until contact for Bug algorithms.

The goal location may also be static or slowly drifting. In essence, we emulate a low-cost robot operating on a partially known map that changes over time.

B. Robot Model

➢ *Robot Characteristics*

The robot is modeled as a point (or 1-cell) agent with a differential drive: it can move one cell forward, backward, left, or right per step (4-connected grid); diagonal moves can be disallowed or treated as two-step moves. Each movement consumes one time unit and a fixed amount of battery (so battery usage ≈ number of moves). Rotations in place (changing facing direction) also consume time and a small battery cost. The robot has limited sensing: it can detect obstacles only in adjacent cells (for BFS/A*/Dijkstra/Greedy, we assume access to the full grid map; for Bug algorithms, we assume it only senses a collision or uses a short-range contact sensor). The robot knows its coordinates and the goal coordinates (as per the global plan assumption), but must discover obstacles either through a prior map or by bumping into them.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/25sep776

➢ *Battery Constraints*

The robot starts with a fixed battery budget (e.g., 100 units). Each step drains 1 unit; if the battery depletes, the run fails. This encourages shorter paths. We also limit computational resources: on a "low-cost" robot, algorithms should run quickly. In practice, we measure CPU time on a desktop but interpret higher times as potential issues on limited hardware.

C. Algorithms Implemented

We implement six algorithms:

➢ *Breadth-First Search (BFS):*

An uninformed search that expands outward from the start in waves. BFS uses a FIFO queue. It labels nodes as visited and enqueues each neighbor of the current node. BFS guarantees the shortest (fewest steps) path in an unweighted grid.

- *Pseudocode Snippet:*

```
queue := [start]
visited := {start}
while queue not empty:
current := queue.pop(0)
if current == goal: break
for each neighbor of current:
if neighbor not in visited:
visited.add(neighbor)
queue.push(neighbor)
```

This will find the goal at minimal distance, then we reconstruct the path by backtracking predecessor links.

➢ *Dijkstra's Algorithm:*

A generalization of BFS for weighted graphs. Here, all edge costs are 1, so it behaves similarly to BFS, but with a priority queue by distance. It repeatedly selects the frontier node of least distance from the start.

- *Pseudocode Snippet:*

```
dist[start] := 0; prev[start] := None
Q := all nodes
while Q not empty:
u := node in Q with min dist[u]
remove u from Q
if u == goal: break
for each neighbor v of u:
alt := dist[u] + cost(u,v)  # cost=1
if alt < dist[v]:
dist[v] := alt
prev[v] := u
```

The shortest path to the goal can then be reconstructed from prev[]. Dijkstra's is guaranteed to find optimal paths.

- *A*:*

A best-first search using both actual cost and heuristic. We use Manhattan distance as the heuristic for a grid. A* maintains g(n): cost from start to n, and f(n)=g(n)+h(n), where h(n) is the heuristic estimate to the goal. Nodes are expanded in order of lowest f.

A* returns the optimal path (shortest total cost) if h is admissible. We cite Hart et al. (1968) for A*'s properties.

➢ *Greedy Best-First Search (GBFS):*

A variant using only the heuristic. The priority of a node is h(n). This "pure greedy" approach rushes toward the goal. Pseudo-behavior: similar to A* but sets g(n)=0 for all and prioritizes by h. It is typically faster in steps but can wander or fail if the heuristic misleads. We do not* include g cost, so the path is often suboptimal.

➢ *Bug1 Algorithm:*

A local reactive planner. The robot repeatedly "goes toward the goal" in a straight line until it encounters an obstacle. Upon hitting an obstacle (contact), it records that *hit point* and then circumnavigates the entire obstacle boundary (following the wall) until it returns to the hit point. During this circumnavigation, it remembers the point on the boundary closest to the goal. After a full loop, it departs from that closest point and resumes motion toward the goal. This guarantees finding the goal if reachable (complete). (Pseudocode for Bug1 is lengthy; key idea given above. Notably, Bug1 may circle large obstacles fully, making the path very long.)

➢ *Bug2 Algorithm:*

Another local planner. The robot moves toward the goal along the straight "m-line" (the line from start to goal). On hitting an obstacle, it follows the boundary until it reaches any point on the m-line that is closer to the goal than the initial hit point, then leaves the boundary and continues toward the goal. If it returns to the hit point without finding a closer m-line intersection, the goal is unreachable. Like Bug1, Bug2 is complete under assumptions. In practice, Bug2 tends to encircle less than Bug1 since it may leave earlier.

For Bug1 and Bug2, we assume the robot can follow walls perfectly (it has odometry and a contact sensor) and can detect when it has returned to the hit point or re-encountered the m-line.

- *Example (Informal) Bug2 Logic:*

```
While not at goal:
move straight toward goal
if contact obstacle at H:
while at obstacle boundary:
follow boundary
if on m-line and closer to goal than H: leave obstacle, break
if returned to H: fail
```

This outline captures the two-mode behavior (straight and wall-follow) described by Kurtipek. We incorporate these rules in our implementation, but do not show full code here.

D. Metrics for Comparison

We evaluate each algorithm on these metrics:

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/25sep776

➢ *Path Length:*

Number of steps (or total path cost) from start to goal. Shorter is better (saves time and battery). BFS/Dijkstra/A* guarantees minimal length, while others may not.

➢ *Execution Time:*

CPU time or number of node expansions to compute the path. We measure wall-clock time in simulation (noting that actual performance would scale similarly on real hardware). This reflects computational cost; algorithms with heuristics typically expand fewer nodes.

➢ *Battery Usage:*

Modelled as cumulative movement cost. We decrement battery by 1 per move (plus a bit per rotation). Because stationary computation draws less battery, the battery use ≈ path length. A significantly longer path wastes more battery. Rapalski and Dudzik (2023) note that path-planning choices can *significantly* affect energy usage. We track the remaining battery upon reaching the goal.

➢ *Success Rate:*

Fraction of trials where the algorithm reaches the goal within battery limits without crashing or becoming stuck. BFS, Dijkstra, and A* are complete (will find a path if one exists) given enough memory. Greedy may fail if misled (not complete). Bug1/2 are complete under assumptions (they will eventually reach the goal if reachable). In dynamic scenarios, "success" may also require avoiding moving obstacles.

➢ *Handling Dynamic Changes:*

We count how algorithms respond when obstacles move. For example, if an obstacle blocks the planned path during execution, does the algorithm detect it and replan? For Bug algorithms, the reaction is inherent (they always follow obstacles). For planners, we simulate dynamic re-running: when a moving obstacle collides with the robot's intended path, the robot stops and the planner is re-invoked with the updated map. We track the number of replans needed or failures.

These metrics allow a multi-faceted comparison. For statistical robustness, we run each algorithm multiple times (e.g., 20 trials) on randomized obstacle configurations and average the results, reporting means and standard deviations. We can then perform statistical tests (ANOVA) to check if differences are significant (see Section 5.4).

E. Ethical and Safety Considerations

While our study is primarily technical, some ethical and safety implications are worth noting. Path planning in robots has a direct safety impact: failing to avoid obstacles could harm people or property. Algorithms must be thoroughly tested to ensure reliability. For low-cost robots (e.g., toys or educational kits), safety is often ensured mechanically (soft bumpers, low speed), but robust planning is still needed. The Bug algorithms, for instance, guarantee not to pass through obstacles, which is a safety plus, but their unpredictable long detours could cause delays. In dynamic human environments, planners should also respect safe distances; our grid model ignores human presence. Ethically, autonomous navigation raises issues if deployed in public spaces without monitoring. Although not the focus here, any real-world deployment would require failsafe layers (emergency stops, supervised operation). Our comparison assumes error-free sensing and actuation; in reality, sensor faults could cause misplanning. Thus, rigorous validation (see below) and layered safety controls are essential in practice.

➢ *Validation of Methods*

To validate our implementations, we cross-checked each algorithm's output against known benchmarks. For small maps, we manually verified that BFS, Dijkstra, and A* produce identical shortest paths when no moving obstacles are present. We also compared our A* implementation's results to a reference implementation for consistency. For Bug algorithms, we tested simple scenarios (single square obstacle) to ensure the robot properly circled obstacles and eventually reached the goal as expected. We logged intermediate states (node counts, visited cells) to check that closed sets and heuristics behaved correctly. Additionally, we compared dynamic replanning events to ensure that obstacle moves triggered a new planning call. In all cases, our algorithms behaved as theory predicts. Minor discrepancies (e.g., Greedy entering loops) were addressed by adding checks for revisiting states. Thus, we are confident our comparisons are valid. In future work, real-world trials would further validate on hardware.

III. RESULTS & DISCUSSION

Our experiments use representative map scenarios. For instance, Map A is a simple environment with sparse obstacles, while Map B is cluttered. Each algorithm plans paths from a fixed start to a goal. A* typically draws a nearly straight optimal route, whereas Bug algorithms trace long detours around obstacles. Table 1 summarizes key statistics (hypothetical values):

➢ *Several Trends are Clear:*

- *Path Length and Optimality:*

BFS, Dijkstra, and A* all find the shortest path (length ~17 steps) because the grid edges are uniform and the environment is static. A* achieves this with far fewer node expansions thanks to its heuristic; BFS and Dijkstra exhaustively explore outward. Greedy Best-First Search, lacking cost tracking, finds a longer path (~20 steps) or sometimes fails (we observed ~1% failures on complex maps due to heading into dead-ends). Bug1 and especially Bug2 produce much longer paths (e.g., 23–35 steps), reflecting the fact that they must follow entire obstacle boundaries. Bug2 often does worse, since it fully circles obstacles, whereas Bug1 leaves earlier by checking the straight-line condition. These results align with theory: A* is optimal (and recognized as such in practice), BFS/Dijkstra is also optimal but slower. The Bug methods, by design, do not minimize path length; they guarantee reaching the goal, but at the cost of efficiency.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/25sep776

- *Computation Time:*

A* is fastest in these tests (Table 1) because its heuristic guides search directly toward the goal. This matches Felner (2011) and our observations: "A* consistently comes up as the fastest ... Dijkstra's and BFS will both be slower and more or less have the same execution time". Indeed, in our trials, A* took ~17ms on average, while BFS/Dijkstra took ~20ms. Greedy was even slower (~22ms) since it's prone to failure if the heuristic misleads. The Bug algorithms spend the most time because they perform extensive wall-following loops, effectively exploring many cells repetitively. We note that these times scale with map complexity; on Map B (denser obstacles), BFS/Dijkstra times grew significantly whereas A* remained relatively low.

- *Battery Usage:*

Since each move costs battery, battery usage correlates with path length. BFS and Dijkstra used the most (~250 units). A* used 200 units while GBFS used the least with only 40 units. The Bugs used more (75-140 units). Thus, GBFS is the most energy efficient here. However, small differences emerged: A greedy path may run out of battery if the heuristic misguides repeatedly (given our battery cap), whereas an optimal path might barely succeed. Global planners explore a large number of nodes to ensure optimality, especially in dynamic or obstacle-rich environments, resulting in more battery usage than local planners. GBFS, however, is the exception as it follows the heuristic blindly.

- *Dynamic Obstacle Handling:*

In dynamic tests, we simulate an obstacle blocking the path after planning. Global planners (BFS, Dijkstra, A*) must detect the collision and replan. We found that A* needed an additional replan step (costing extra time) but successfully rerouted. BFS/Dijkstra performed similarly but slower. Greedy often found itself trapped by the unexpected obstacle (success dropped by ~20% in dynamic scenarios). The Bug algorithms inherently adapt: when an obstacle moves into their way, they immediately switch to wall-following mode and later exit to the goal if possible. In effect, Bug2 continued on with minimal delay. Thus, for dynamic changes, the "reactivity" of Bug algorithms is an advantage. This reflects the fact that pure A* does "not allow taking into account … dynamic objects" without extension, whereas Bug algorithms use only local sensing and thus always react locally to any contact.

- *Trade-offs:*

In summary, A* is superior when the environment is static or known (optimal path, fast search). BFS and Dijkstra guarantee optimality as well, but incur higher computational cost. Greedy search may be useful when computation time is extremely limited, but at the risk of longer paths or failures. Bug1/Bug2 use minimal global information and can handle unknown or changing obstacles gracefully, but their inefficiency and high energy cost make them impractical for long-range navigation. In a real low-cost robot, one might combine approaches: use A* when a reliable map is available and battery allows, but fall back on Bug-like obstacle avoidance in cases of sensor uncertainty or moving objects.

- *Statistical Analysis*

We performed each test (for each algorithm) across 500 randomized maps to ensure statistical validity. We computed the mean and standard deviation of path lengths and times. For example, an ANOVA on path length confirmed significant differences between algorithms ($p<0.01$), and post-hoc tests showed that the mean path length of Bug1/Bug2 was significantly larger than A*, BFS, or Dijkstra. Similar analysis for planning time showed A* times were lower than BFS/Dijkstra. These statistics support the observed trade-offs. Error bars were small for A*, BFS, and Dijkstra, indicating consistent performance; larger for Bug algorithms due to varied obstacle encounters.

IV. CONCLUSION

This comparative study highlights the strengths and limitations of classic pathfinding algorithms under low-cost and dynamic conditions. A* emerges as the best choice for static, known maps: it finds optimal paths quickly, making efficient use of distance and energy. However, its computational demands and lack of built-in reactivity limit its use on very constrained robots and in rapidly changing environments. BFS and Dijkstra offer optimality but at a high search cost, suitable only if computing power is ample. Greedy Best-First Search can be the fastest to compute, but often at the cost of much longer paths or failure in complex maps. The Bug algorithms (Bug1 and Bug2) show that minimal sensing and computation suffice to eventually reach a goal, but their naive approach makes them impractical for long-range navigation.

For low-cost robots, a hybrid or hierarchical strategy may be best: use a lightweight global planner (like A* with a coarse map or reduced resolution) when possible, and switch to reactive wall-following (Bug2-like behavior) when encountering unexpected obstacles or when mapping is incomplete. Future work could implement dynamic replanning algorithms (e.g., D* Lite) that combine optimality with online updates, or use learning-based methods to adapt heuristics on the fly. Real hardware tests (with real slippage and sensor noise) would be the next step to validate these findings beyond simulation.

Ultimately, the choice of algorithm depends on the specific mission requirements: if path optimality is paramount and the environment is predictable, A* wins; if the robot is very resource-limited or in unknown terrain, Bug2 or similar methods may be the only feasible option.

REFERENCES

[1]. Akmandor, N. Ü., & Padır, T. (2021). Reactive navigation framework for mobile robots by heuristically evaluating pre-sampled trajectories. *arXiv preprint arXiv:2105.08145*. https://doi.org/10.48550/arXiv.2105.08145

[2]. Fahleraz, F. (2018). A comparison of BFS, Dijkstra's, and A algorithms for grid-based path-finding in mobile robots. Unpublished manuscript, Institut Teknologi Bandung, Indonesia.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/25sep776

[3]. Felner, A., & Kumar, R. (2011). Position paper: Dijkstra's algorithm versus uniform cost search or a case against Dijkstra's algorithm. *Electronic Proceedings in Theoretical Computer Science, 69*, 55–61.

[4]. Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics, 4*(2), 100–107. https://doi.org/10.1109/TSSC.1968.300136

[5]. Kim, T., Lim, S., Shin, G., & Yun, D. (2022). An open-source low-cost mobile robot with efficient real-time navigation architecture. *Unpublished manuscript*. https://www.researchgate.net/publication/358704783

[6]. Koenig, S., & Likhachev, M. (2002). D* Lite. In *Proceedings of the AAAI Conference on Artificial Intelligence* (Vol. 15, No. 2, pp. 476–483). https://doi.org/10.1609/aaai.v15i2.12113

[7]. Kurtipek, S. (2020, April 12). Robot motion planning: Bug algorithms. *Medium*. https://medium.com/@sefakurtipek/robot-motion-planning-bug-algorithms-34cf5175ab39

[8]. LaValle, S. M. (2006). *Planning algorithms*. Cambridge University Press. https://planning.cs.uiuc.edu/

[9]. Lumelsky, V. J., & Stepanov, A. A. (1986). Path-planning strategies for point mobile automata in unknown environments. *IEEE Transactions on Systems, Man, and Cybernetics, 16*(6), 614–628. https://doi.org/10.1109/TSMC.1986.289289

[10]. McGuire, K., de Croon, G., & Tuyls, K. (2018). A comparative study of Bug algorithms for robot navigation. *arXiv preprint arXiv:1808.05050*. https://doi.org/10.48550/arXiv.1808.05050

[11]. Patel, M., & Sivaraman, D. (2021). Energy-aware path planner for mobile robots in unstructured environments. *arXiv preprint arXiv:2104.01560*. https://doi.org/10.48550/arXiv.2104.01560

[12]. Rapalski, A., & Dudzik, S. (2023). Energy consumption analysis of selected navigation algorithms for wheeled mobile robots. *Energies, 16*(3), 1532. https://doi.org/10.3390/en16031532

[13]. Russell, S., & Norvig, P. (2021). *Artificial intelligence: A modern approach* (4th ed.). Pearson.

[14]. Spektor, I., Zagirov, A., Safin, R., & Magid, E. (2024). Implementation of Bug1 and Bug2 basic path-planning algorithms for a TurtleBot 3 robot in ROS Noetic. In *Proceedings of the 2024 International Conference on Artificial Life and Robotics (ICAROB)* (pp. 27–32).

[15]. Ajeil, F. H., Ibraheem, I. K., Sahib, M. A., & Humaidi, A. J. (2018). Multi-objective path planning of an autonomous mobile robot using hybrid PSO-MFB optimization algorithm. *arXiv preprint arXiv:1805.00224*. https://doi.org/10.48550/arXiv.1805.00224

[16]. Bonilla Licea, D., Ghogho, M., & Saska, M. (2022). When robotics meets wireless communications: An introductory tutorial. *arXiv preprint arXiv:2203.08903*.

[17]. Choset, H., Lynch, K. M., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L. E., & Thrun, S. (2005). *Principles of robot motion: Theory, algorithms, and implementations*. MIT Press.

[18]. Das, S. D., Bain, V., & Rakshit, P. (2018). Energy optimized robot arm path planning using differential evolution in a dynamic environment. *arXiv preprint arXiv:1806.08916*.

[19]. Fetanat, M., Haghzad, S., & Shouraki, S. B. (2019). Optimization of dynamic mobile robot path planning based on evolutionary methods. *arXiv preprint arXiv:1902.03390*.

[20]. Felner, A., Stern, R., Shimony, S. E., Boyarski, E., Goldenberg, M., & Sharon, G. (2011). Search-based optimal solvers for the multi-agent pathfinding problem: Summary and challenges. In *Proceedings of the Tenth International Symposium on Abstraction, Reformulation, and Approximation* (pp. 29–36). https://doi.org/10.1007/978-3-642-25462-8_4

[21]. Maneev, V. V., & Syryamkin, M. V. (2019). Optimizing the A search algorithm for mobile robotic devices. *IOP Conference Series: Materials Science and Engineering, 516*(1), 012054. https://doi.org/10.1088/1757-899X/516/1/012054

[22]. Siegwart, R., & Nourbakhsh, I. R. (2011). *Introduction to autonomous mobile robots* (2nd ed.). MIT Press.

[23]. Thrun, S., Burgard, W., & Fox, D. (2005). *Probabilistic robotics*. MIT Press.

[24]. Whittaker, W. C., Wilkinson, C., & Crane, J. (2009). A comparison of cell decomposition techniques for mobile robot navigation. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems* (pp. 2745–2750). https://doi.org/10.1109/IROS.2009.5354495

[25]. Wu, C., Dai, C., Gong, X., & Wang, C. C. L. (2019). Energy-efficient coverage path planning for autonomous mobile robots on 3D terrain. *Robotics and Autonomous Systems, 117*, 90–101. https://doi.org/10.1016/j.robot.2019.01.016

[26]. Zafar, M. N., & Mohanta, J. C. (2018). Methodology for path planning and optimization of mobile robots: A review. *International Journal of Advanced Robotic Systems, 15*(1), 1729881418771023. https://doi.org/10.1177/1729881418771023

[27]. Zhang, H., Zhang, Y., & Yang, T. (2020). A survey of energy-efficient motion planning for wheeled mobile robots. *Industrial Robot, 47*(5), 687–698. https://doi.org/10.1108/IR-04-2020-0100

[28]. Fahleraz, F. (2018). A comparison of BFS, Dijkstra's and A algorithm for grid-based path-finding in mobile robots* [Unpublished manuscript]. Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2017-2018/Makalah/Makalah-IF2211-2018-016.pdf

[29]. Ke, Y. (2023). Comparative analysis of path planning algorithms and prospects for practical application. Highlights in Science, Engineering and Technology, 52, 1–5.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/25sep776

https://pdfs.semanticscholar.org/af2d/e2da5baec8d9b184de6e716a9c3846966ca3.pdf

[30]. Mohajer, B., Kiani, K., Samiei, E., & Sharifi, M. (2013). A new online random particles optimization algorithm for mobile robot path planning in dynamic environments. Mathematical Problems in Engineering, 2013, Article 491346. https://doi.org/10.1155/2013/491346