PeerJ Computer Science

Fast and efficient indoor navigation: a hybrid pathfinding approach using rapidly-exploring random tree (RRT)-connect and Dijkstra's algorithm

Ramamoorthy Sriramulu*, Abhishek Yadav and Subha Bal Pal*

Department of Computing Technologies, School of Computing, SRM Institute of Science and Technology, Kattankulathur, Tamil Nadu, India

* These authors contributed equally to this work.

ABSTRACT

This article introduces a hybrid approach to enhance indoor pathfinding and navigation within complex multistory environments by integrating rapidly-exploring random tree (RRT)-Connect and Dijkstra's algorithm. We propose a novel solution leveraging the strengths of RRT-connect for rapid path generation, combined with Dijkstra's algorithm for refining and optimizing the final route. Our method leverages the rapid exploration of RRT—Connect while refining paths using Dijkstra's algorithm, resulting in fewer nodes explored compared to Lazy Theta* while maintaining efficiency. Experimental results demonstrate that our hybrid approach significantly reduces computational overhead, with RRT-Connect exploring approximately 1,750 nodes—outperforming RRT (2,000 nodes), RRT* (1,850 nodes), and Dijkstra (1,780 nodes). The algorithm achieves up to 50% faster execution in narrow spaces compared to traditional RRT, making it well-suited for real-time navigation. Additionally, parallel processing optimizes performance, ensuring efficient pathfinding in dynamic environments. A Next.js-based frontend visualization system further enhances usability by rendering path nodes in real time. This hybrid approach balances rapid exploration, optimal path computation, and computational efficiency, making it a robust solution for indoor navigation in large-scale and complex environments.

Subjects Adaptive and Self-Organizing Systems, Algorithms and Analysis of Algorithms, Autonomous Systems, Optimization Theory and Computation, Spatial and Geographic Information Systems

Keywords Hybrid RRT-connect Dijkstra's algorithm, Indoor navigation, Multistory buildings, RRT-connect, Pathfinding, Dijkstra's algorithm

Submitted 14 November 2024
Accepted 22 June 2025
Published 14 October 2025

Corresponding authors
Ramamoorthy Sriramulu, ramamoos@srmist.edu.in
Abhishek Yadav, as2165@srmist.edu.in

Academic editor
Valentina Emilia Balas

Additional Information and Declarations can be found on page 34

DOI **10.7717/peerj-cs.3028**

INTRODUCTION

Efficient and accurate pathfinding has become a critical area of focus, especially in environments where complex structures such as multistory buildings exist. From college campuses to large shopping malls and smart cities, there is an increasing need for users to navigate these environments seamlessly. Traditional navigation systems and applications, which rely primarily on 2D projections, face significant challenges when applied to environments with multiple levels, obstacles, and dynamic conditions. Furthermore, the

demand for real-time navigation and minimal computational delays adds another layer of complexity to these systems (*Liu et al., 2020*).

Traditional algorithms such as Dijkstra's and A* have been effective in pathfinding but struggle in dynamic, multilevel environments due to their computational overhead and inefficiencies in large-scale maps (*Singh et al., 2024*). While Dijkstra's algorithm ensures optimality, its efficiency degrades significantly in high-dimensional spaces or environments with complex layouts. This is due to its exhaustive nature, which leads to substantial computational overhead in large graphs, especially when dealing with multiple levels. This limitation makes it unsuitable for real-time pathfinding in large multistory buildings where nodes represent different spatial locations across floors, and dynamic changes such as obstacles and re-routing are frequent (*Zhang et al., 2021*).

To address these limitations, this article proposes the use of rapidly-exploring random trees (RRT), particularly the RRT-Connect variants, for spatial navigation in multilevel environments. RRT algorithms are considered highly effective for high-dimensional spaces due to their ability to focus on feasible paths without the need for an exhaustive search of the entire space. They explore randomly and rapidly, extending the search tree towards a goal while avoiding unnecessary exploration. RRT-Connect enhances the basic RRT algorithm by growing trees from both the start and goal configurations, eventually connecting them for a more efficient solution (*Faramondi et al., 2014*; *Zhou et al., 2022*).

In addition to algorithmic improvements, the system integrates parallel processing capabilities, taking advantage of CPU parallelization. The implementation of parallelization reduces the time complexity associated with expanding trees and finding optimal paths, especially in large, dynamic environments. This allows for real-time computation, making it viable for use in environments where fast responses are crucial, such as emergencies or navigation in crowded spaces (*El-Sheimy & Li, 2021*).

In summary, this research introduces a spatial database for multilevel navigation that leverages RRT-based algorithms and parallel processing to overcome the shortcomings of traditional approaches. The system is designed to provide users with an efficient, real-time solution for navigating complex environments, without the computational limitations of exhaustive search algorithms like Dijkstra.

LITERATURE OVERVIEW

The field of pathfinding algorithms has evolved significantly over the years, with each algorithm designed to solve specific challenges in navigation and route optimization. The increasing complexity of real-world environments, such as multilevel structures and dynamic obstacles (*Tran & Ha, 2022*), has driven the development of more efficient algorithms, particularly in scenarios where real-time computation is essential. In this literature review, we compare traditional algorithms like Dijkstra's and A*, as well as more methods like RRT and their variants, in the context of spatial databases and multilevel navigation.

Traditional pathfinding algorithms

Dijikstra's algorithm

Dijkstra's algorithm is one of the most well-known and widely used algorithms for finding the shortest path between two points on a graph. It guarantees finding the optimal solution by exhaustively exploring every possible path until the shortest one is determined. It has been applied extensively in 2D environments, such as road networks and basic indoor navigation.

However, its exhaustive nature becomes a limitation when applied to more complex, high-dimensional environments. In multistory buildings or environments with dynamic obstacles, Dijkstra's algorithm suffers from excessive computation time and memory consumption. Furthermore, since the algorithm explores all nodes, it struggles with real-time efficiency in scenarios where fast responses are critical.

A algorithm*

A* builds upon Dijkstra's algorithm by introducing a heuristic function to guide the search process more efficiently. It reduces computational overhead by focusing on the most promising nodes, balancing exploration and path cost. A* is faster than Dijkstra in practice for most applications and has been applied in many navigation systems, particularly where obstacles are sparse and the environment is mostly static.

However, like Dijkstra's algorithm, A* faces challenges in environments with multiple levels and dynamic obstacles, particularly when the heuristic used is not well-suited to the problem space. Moreover, A* can struggle with memory efficiency in large environments due to its reliance on storing the open and closed lists.

Modern pathfinding algorithms

Rapidly-exploring random trees (Algorithm 1)

RRT is a more recent algorithm designed to tackle the shortcomings of traditional pathfinding methods in high-dimensional and dynamic spaces like robots or autonomous driving path findings. RRT works by incrementally building a search tree by randomly sampling the environment, expanding the tree toward each sample. Unlike Dijkstra and A*, RRT does not attempt to explore the entire graph exhaustively but focuses on expanding the tree based on random exploration. Furthermore, In Dijkstra's algorithm, the graph is predefined and fully explored, with edges representing the known connections between nodes. Meanwhile, In RRT the graph is built incrementally by adding random samples as new nodes and connecting them to the nearest existing nodes in the tree. As shown in Fig. 1. It is exploratory and adapts based on the configuration space without needing the entire graph structure beforehand (*Martinez, Jacinto & Montiel, 2023*).

The basic steps are as follows:

1. It constructs a random tree at the starting point X_{init} of the two-dimensional state space as the root node;
2. A random sampling point X_{rand} is generated in the free search space and used to guide the expansion of the random tree;

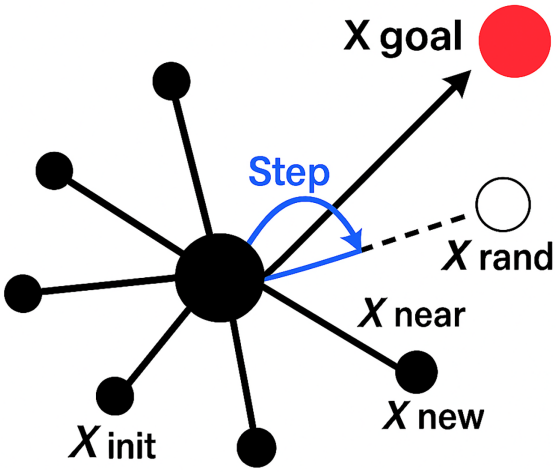

Figure 1 **Visualization of the rapidly-exploring random tree (RRT) algorithm.**
Full-size DOI: 10.7717/peerj-cs.3028/fig-1

3. After traversing the nodes that have been generated in the whole random tree, the tree node that is closest to the random point X_{rand} is found, selected, and defined as X_{near};
4. From the node X_{near}, along the extension direction of the node X_{rand} as the extension direction, the appropriate step size is expanded, and the appropriate step size is set as the branch length to generate a new node X_{new} as the new tree node;
5. If an obstacle is encountered in the expansion process, the expansion is canceled, and sampling is performed again. The path repeats the above iterative process until the target node exceeds the specified number of iterations and finally forms a fast-expanding random tree path, ending the search.

Wang et al. (2023), *Brad & Dolha (2021)*.

RRT has become a popular choice in robotics, automated navigation, and multilevel environments due to its ability to quickly generate paths in complex spaces. However, the basic RRT algorithm does not guarantee an optimal solution and can suffer from slow convergence in certain scenarios.

RRT-Connect and bi-directional RRT

RRT-Connect is an extension of the RRT algorithm that attempts to improve the convergence speed by growing two trees simultaneously: one from the start point and one from the goal. The trees explore the environment independently but attempt to connect to each other as they grow, significantly reducing the time needed to find a feasible path. Bi-directional RRT works similarly, expanding the search from both the start and goal nodes simultaneously but is more efficient in larger environments where growing a single tree takes too long (*Fan et al., 2024*).

Algorithm 1 **RRT algorithm (Rapidly-exploring random tree).**

Input: M, x_{init}, x_{goal}
Output: A path τ from x_{init} to x_{goal}

```
1  1: τ.init();
2  2: for i ← 1 to n do
3        3:    x_rand ← Sample(M);
4        4:    x_near ← Near(x_rand, τ);
5        5:    x_new ← Steer(x_rand, x_near, StepSize);
6        6:    E_t ← Edge(x_new, x_near);
7        7:    if CollisionFree(M, E_t) then
8              8:    τ.addNode(x_new);
9              9:    τ.addEdge(E_t);
10       end
11       10:   if x_new == x_goal then
12             11:   Success();
13       end
14 end
```

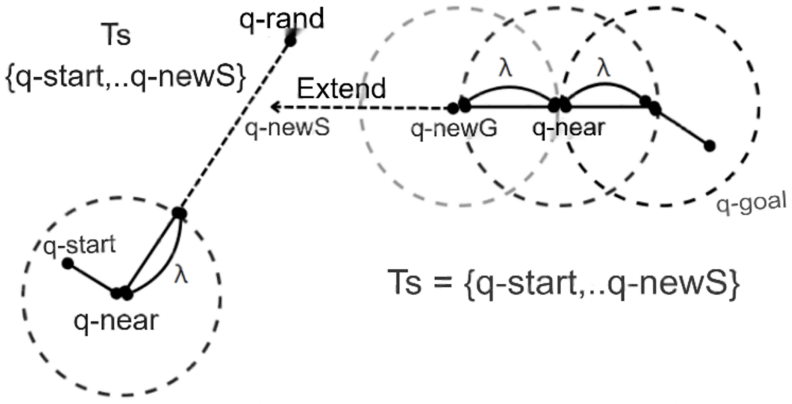

Figure 2 **RRT-Connect algorithm: extend from tree Tb with root as goal position q-goal to tree Ta with root as start position q-start.** Full-size DOI: 10.7717/peerj-cs.3028/fig-2

As shown in Fig. 2. The RRT-Connect algorithm grows two trees: one from the start position q_{start} and the other from the goal position q_{goal}. The goal is to connect the two trees to find a collision-free path.

The RRT algorithm attempts to find a path by sampling random configurations in the space and then extending the tree to include these samples. The tree grows from an initial point q_{start}, and in each iteration, a random configuration q_{rand} is generated. The algorithm then extends the tree by moving a step from the nearest node q_{near} towards q_{rand}, but stopping if a collision occurs.

The extension function is given as:

$$q_{\text{new}} = q_{\text{near}} + \varepsilon \frac{(q_{\text{rand}} - q_{\text{near}})}{\|q_{\text{rand}} - q_{\text{near}}\|} \tag{1}$$

where:

- q_{new} is the new node,
- ε is the step size,
- q_{rand} is the random configuration, and
- q_{near} is the nearest node in the tree to q_{rand}.

RRT-Connect extends this by simultaneously growing two trees: one from the start and one from the goal. The trees will attempt to connect to each other as they grow. In each iteration, the algorithm grows tree T_a from q_{start} towards the random configuration q_{rand}, and then grows tree T_b from q_{goal} towards q_{new}, the new node in tree T_a.

The extension of tree T_b towards tree T_a is done using the same extension function as Eq. (1), but with the trees reversed:

$$q'_{\text{new}} = q'_{\text{near}} + \varepsilon \frac{(q_{\text{new}} - q'_{\text{near}})}{\|q_{\text{new}} - q'_{\text{near}}\|} \tag{2}$$

where:

- q'_{new} is the new node added to tree T_b,
- q'_{near} is the nearest node in tree T_b to q_{new}, and
- q_{new} is the new node generated in tree T_a.

If q_{new} from tree T_a and q'_{new} from tree T_b are close enough, the trees are connected, and a feasible path has been found.

In mathematical terms, the two trees are connected when:

$$\|q_{\text{new}} - q'_{\text{new}}\| < \delta \tag{3}$$

where δ is a predefined threshold for connection.

The computational complexity of RRT-Connect depends on the number of nodes generated in both trees, which is often improved by utilizing parallel processing techniques such as CPU parallelization (*Hidalgo-Paniagua et al., 2018*).

These algorithms are well-suited for real-time applications in dynamic environments, where traditional algorithms like Dijkstra and A* are too slow. Furthermore, by leveraging parallel processing such as CPU parallelization the performance of these algorithms can be significantly improved, enabling their application in real-time scenarios like complex indoor navigation.

Finally, overall overview of pathfinding algorithms is summed up, as shown in Fig. 3.

Proposed spatial databases in navigation

Spatial databases are a critical component of any navigation system. They store information about the environment, such as the coordinates of nodes (*e.g.*, rooms, floors)

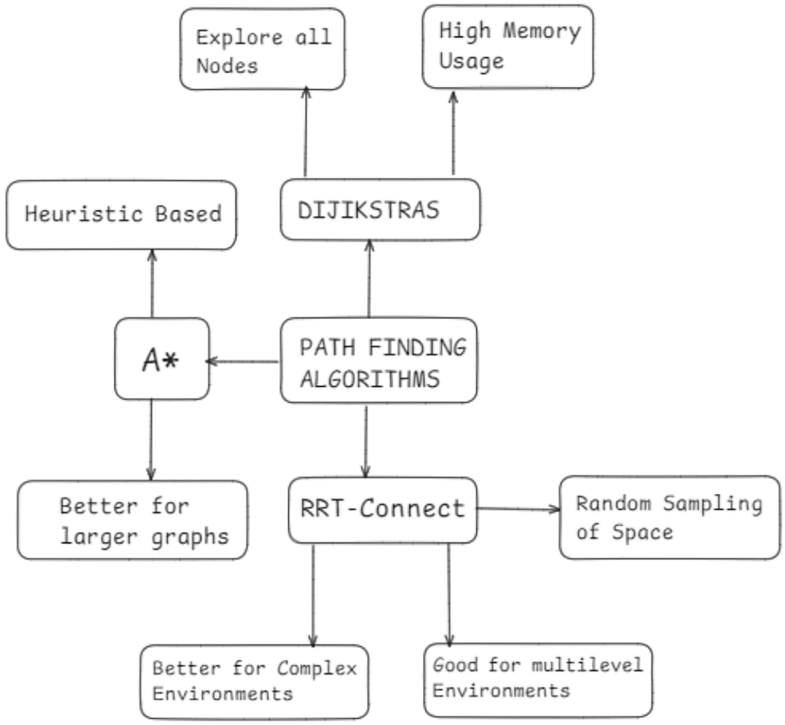

Figure 3 **Overview of pathfinding algorithms and key characteristics.**
Full-size DOI: 10.7717/peerj-cs.3028/fig-3

and the paths between them. The efficiency of a pathfinding algorithm often depends on the underlying data structure of the spatial database.

Traditional navigation systems rely on simple relational databases to store nodes and paths in a 2D plane. However, as environments grow in complexity (*e.g.*, multilevel buildings), these databases must store spatial information in multiple dimensions and manage dynamic updates to node locations or paths as obstacles and conditions change. The efficiency of querying and updating the database directly impacts the performance of pathfinding algorithms.

Modern spatial databases integrate advanced indexing techniques such as R-trees or quadtree structures, enabling faster queries in multidimensional spaces. These databases are essential for supporting algorithms like RRT-Connect, which rely on rapid, frequent lookups of node and path data during the tree expansion process (*Kanth, Ravada & Abugov, 2002*).

In addition to advanced indexing, modern spatial databases often incorporate real-time data processing capabilities, allowing them to handle dynamic changes in the environment, such as moving obstacles or evolving terrain conditions. This real-time adaptability is crucial for algorithms like RRT-Connect, which require the system to continuously assess the feasibility of paths and re-route as necessary. By leveraging these spatial databases, the pathfinding algorithms can maintain high efficiency and accuracy even in complex, ever-changing environments.

Table 1 Comparison of pathfinding algorithms.

	Algorithm	Optimality	Time complexity	Memory efficiency
1	Dijkstra	Guaranteed	High $O(V^2)$	Low
2	A*	Near-optimal	High $O(V \log V)$	Medium
3	RRT	Non-optimal	Low	High
4	RRT-connect	Near-optimal	Moderate	High

Algorithm comparison and discussion

Based on the research, we can summarize the key features, benefits, and drawbacks of the main pathfinding algorithms in the following Table 1.

PROPOSED METHODOLOGY

In this section, we present a detailed description of the proposed solution, which addresses the shortcomings of traditional pathfinding algorithms like Dijkstra and A* in multilevel, complex environments. Our solution leverages modern pathfinding algorithms such as RRT-Connect, combined with parallel processing techniques, to ensure efficient and real-time navigation in dynamic environments like multistory buildings. This approach also integrates advanced spatial databases to handle dynamic changes in the environment, ensuring that nodes and paths are updated and queried efficiently.

Problem recap

Traditional algorithms like Dijkstra's and A* have been widely used for pathfinding, but they encounter significant challenges when applied to dynamic, multilevel environments. These challenges include:

Excessive computation time

Dijkstra and A* tend to explore nodes exhaustively, which becomes computationally expensive in large graphs.

Inability to scale

As the number of nodes and levels increase (such as in multistory buildings), these algorithms struggle with efficiency and memory usage.

Lack of adaptability to dynamic environments

When dealing with environments that may change frequently (such as obstacles appearing or disappearing), traditional algorithms face challenges in re-computing paths efficiently.

Key components of the proposed solution

The proposed solution is designed to overcome these limitations by incorporating three key innovations:

1. Algorithmic Improvements with RRT-Connect
2. Parallel Processing for Speed and Efficiency
3. Efficient Spatial Database Design

Algorithmic improvements: RRT-connect

RRT-Connect is an extension of the basic RRT algorithm. These algorithms are particularly suitable for complex environments where exhaustive searches like Dijkstra and A* are too slow. The key principle behind RRT-Connect is its rapid exploration of space through random sampling and tree expansion, ensuring that the algorithm does not explore unnecessary paths (*Faramondi et al., 2014*; *Zhou et al., 2022*).

1. **RRT-Connect algorithm**

The RRT-Connect algorithm works by building two trees:

(a) **Start tree:** Grows from the initial point.
(b) **Goal tree:** Grows from the destination point.

The two trees expand toward random samples of the environment, and when they get close enough, they connect, forming a path. The main advantage of RRT-Connect over basic RRT is its dual growth from both the start and goal points, which reduces exploration time and increases the chances of the two trees meeting sooner.

The algorithm proceeds as follows:

(a) **Initialization:** The trees are initialized from the start and goal points.
(b) **Sampling:** A random point is sampled in the environment.
(c) **Tree expansion:** Both trees are extended toward the sampled point.
(d) **Connection check:** The algorithm checks if the trees have been connected.
(e) **Path construction:** Once the trees connect, the complete path is constructed by combining the two trees.

More in-detailed explanation is provided through a flowchart in Fig. 4.

2. **Path construction**

RRT-Connect is a variant of the RRT algorithm that improves pathfinding efficiency by using two trees: one grown from the start node and another from the goal node. Unlike standard RRT, where the trees grow independently, RRT-Connect actively attempts to connect the two trees as they expand. This connection mechanism significantly speeds up the process of finding a path between the start and goal nodes.

(a) Each tree grows by sampling random points in the space and extending toward these points.
(b) When a new node is added to one tree, RRT-Connect immediately tries to extend the other tree toward this new node.
(c) This process continues until the two trees connect, forming a complete path. The iterative connection attempts reduce unnecessary exploration, making the algorithm more efficient for environments with obstacles and complex routes.

RRT-Connect's strategy of growing trees toward each other reduces the number of iterations needed to find a solution, particularly in large, multilevel environments. This

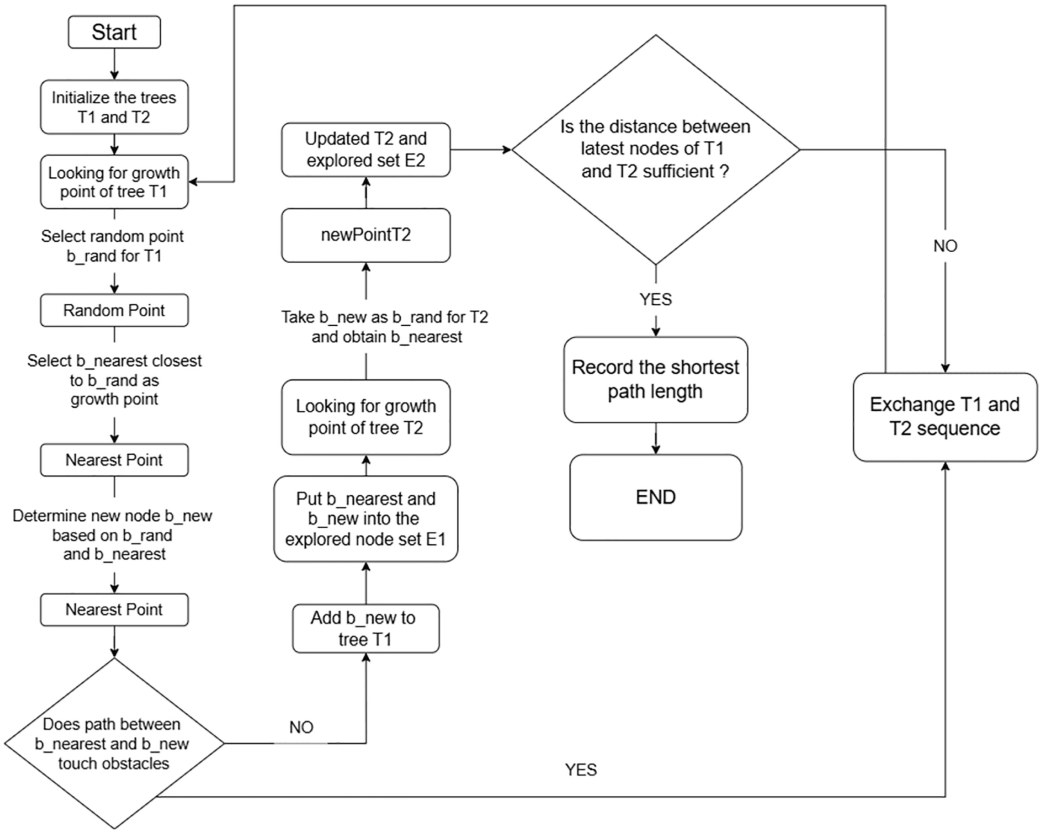

Figure 4 **Flowchart of RRT-connect algorithm.** Full-size DOI: 10.7717/peerj-cs.3028/fig-4

makes it well-suited for indoor navigation systems where the search space can be highly complex and multi-dimensional.

Parallel processing for speed and efficiency

While RRT-Connect improves pathfinding performance, large environments with dynamic changes still pose a challenge due to the number of nodes and paths that need to be explored in real-time. To address this issue, we propose parallel processing techniques that leverage CPU resources to accelerate the algorithm's performance.

CPU parallelization

In environments where real-time performance is critical, the RRT-Connect algorithm can be parallelized to distribute the task of node expansion across multiple CPU cores. Each core handles the expansion of a section of the tree, performing operations such as:

- Sampling a random node
- Checking for obstacles
- Extending the tree toward the node

By distributing these tasks across multiple CPU cores, we can significantly reduce the time taken for the algorithm to find a valid path.

This allows for rapid node exploration and tree growth, particularly in environments where fast response times are required, such as real-time navigation in crowded spaces.

Proposed efficient spatial database design

To support real-time navigation, an efficient spatial database is critical for storing and retrieving information about nodes, paths, and obstacles. The spatial database in our proposed solution stores nodes (representing points of interest such as rooms, floors, or other locations) and the paths between them.

Node storage

Each node in the spatial database is represented by its coordinates (X, Y, Z for 3D spaces, or just X, Y for 2D environments). In a multilevel building, for example, nodes could represent:

- Rooms on different floors
- Staircases or elevators connecting floors
- Hallways or corridors

Path storage

Paths between nodes represent the edges in the graph, with each path having an associated distance and other attributes, such as:

- The time it takes to traverse the path
- The conditions of the path (*e.g.*, obstacles, blocked routes)

The database is structured to allow for fast querying and updating of nodes and paths, particularly in environments where the conditions may change dynamically.

Database query optimization

To ensure efficient querying, the spatial database employs advanced indexing techniques such as:

- **R-trees:** Used to index multi-dimensional data, allowing for efficient spatial queries.
- **Quadtree structures:** Help partition the space into manageable sections, enabling faster lookup of nodes and paths.

By optimizing the database for fast lookups and updates, the system can handle dynamic changes in the environment, such as new obstacles appearing, in real-time without compromising on performance.

Dynamic path recalculation

One of the core features of the proposed solution is its ability to handle dynamic changes in the environment. In real-world navigation scenarios, such as emergencies or changes in building layouts, it is crucial that the system can quickly re-compute paths based on new conditions.

Obstacle handling

The system detects obstacles (*e.g.*, blocked hallways, staircases, or elevators) and updates the database to mark these paths as unavailable. The pathfinding algorithm then recalculates the optimal route based on the updated environment.

Real-time path updates

When a dynamic change is detected, such as a new obstacle in the path, the algorithm uses the information stored in the database to recalculate the best path in real-time, ensuring that users are always provided with the most efficient route.

Summary of the proposed solution

The proposed solution combines the following key components to achieve efficient, real-time navigation in complex environments:

1. **RRT-Connect:**

Modern pathfinding algorithms are designed to handle complex, multilevel environments and generate all routes across obstacles.

2. **Parallel processing:**

Leveraging CPU parallelization to speed up pathfinding, particularly in real-time, dynamic environments.

3. **Dijkstra's algorithm:**

Generated routes can be saved in the format of a pickle file, and then at the time of finding the optimal path, Dijkstra's algorithm is applied over the generated routes to provide the optimal path to the user.

4. **Efficient spatial database design:**

A robust spatial database that stores nodes, paths, and obstacles, allowing for fast lookups and dynamic updates. The spatial database in our proposed solution stores nodes (representing points of interest such as rooms, floors, or other locations) and the paths between them.

By integrating these components, the proposed system addresses the limitations of traditional pathfinding algorithms and provides a scalable, real-time navigation solution for complex environments.

IMPLEMENTATION

System architecture overview

As shown in Fig. 5, a detailed explanation here:

Backend system

The backend system is the core component of the pathfinding operations, integrating the algorithm and parallel processing to navigate complex indoor environments.

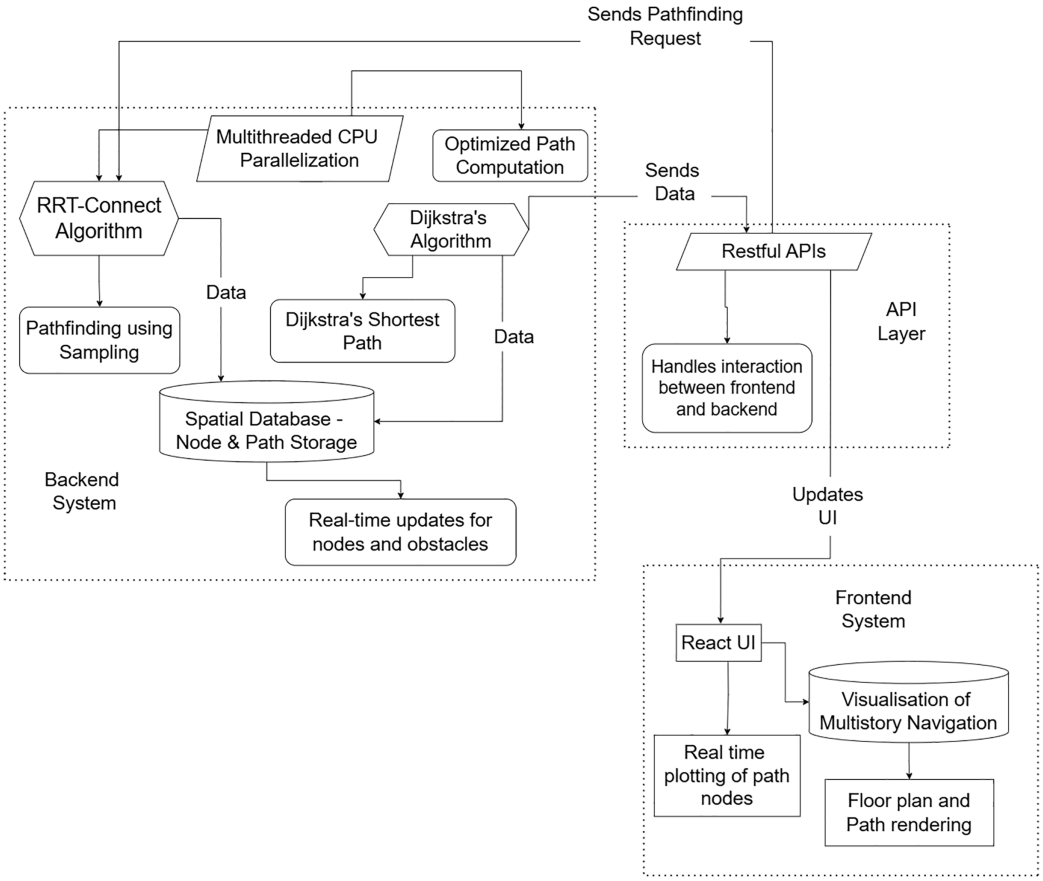

Figure 5 **System architecture of the implementation.** Full-size DOI: 10.7717/peerj-cs.3028/fig-5

Algorithm (RRT-Connect + Dijkstra's)

The algorithm in the proposed solution uses a hybrid approach. The RRT-Connect algorithm rapidly explores random points in the environment to generate potential paths. Once exploration is completed, Dijkstra's algorithm is applied to the explored nodes to compute the optimal and shortest path between the user's start and goal points, ensuring a balance between fast exploration and path optimality (*Zhang et al., 2023*; *Dirik & Kocamaz, 2020*).

Spatial database

The spatial database holds data about the surroundings, containing nodes (representing rooms, halls, *etc.*) and paths (representing connections between these nodes). It is also capable of reflecting real-time updates, such as new obstacles or changes in paths made by external sources, ensuring the algorithms utilize the most current data (*Mohanty & Parhi, 2013*).

Frontend system

The frontend system is built with NextJs to provide a seamless, interactive user interface for real-time navigation updates.

NextJs UI

The user interface is developed using NextJs to enable fast, dynamic updates and real-time path rendering. NextJs efficiently handles the rendering of components, such as the multi-story floor plan, dynamically displaying path nodes and routes based on backend calculations.

The UI takes the data from the user from the floor of the source building to the floor of the destination building. Based upon the Admin settings saved for a floor map base with its respective building name is saved in the database. Upon selection of each building the respective floors are mapped in the options. From further selection of floors, in the building then the output is shown. The floor data, *i.e.*, its number and its building name is saved as a session data in the frontend. Further when the floor view is selected by the user then that respective building floor mapping can be seen. As soon as the button is clicked the information like the floor number, room number, and building name is sent. Then the image base is chosen based on the information sent and on the stored nodes the shortest path is drawn using Dijkstra's algorithm in the backend only; it takes very short time which is then sent in the frontend and is showcased. Much of the data from the application programming interface (APIs) are maintained through session data, if the page gets refreshed then form need to be filled again by the user. For a better example, we selected the source as "Tech Park" and the destination as "University Building". The respective floors and room options were there from which in the source "Floor 2" and "TP 218" is selected. In the Destination "Floor 6" and "UB 612" is selected. The results are shown in Fig. 6.

The floor view of the tech park, as when it is viewed then the information like "Floor 2" and "TP 218" is selected and sent. Based upon this information from "TP 218" to "Lift", a view of 2nd floor is drawn, as shown in Fig. 7.

Further from the "Lift" to the "Entrance Gate" of the ground floor of the tech park is drawn, as shown in Fig. 8.

The floor view of the university building, as when it is viewed then the information like "Floor 6" and "UB 612" is selected and sent. Based upon this information from "UB 612" to "Lift", a view of the 2nd floor is drawn, as shown in Fig. 9.

Further from the "Entrance Gate" to the "Lift" of the ground floor of the university building is drawn, as shown in Fig. 10.

Visualization of multi-story navigation

The frontend allows users to visualize multi-story navigation paths on floor plans. Real-time plotting of path nodes helps users track their progress and adjust routes dynamically when obstacles appear or paths are recalculated. The system highlights explored and unexplored areas to provide users with a clear understanding of their surroundings.

API design

RESTful APIs. The API layer is built using Django (or Flask/Node.js alternatives) to handle requests from the frontend and communicate with the backend for pathfinding

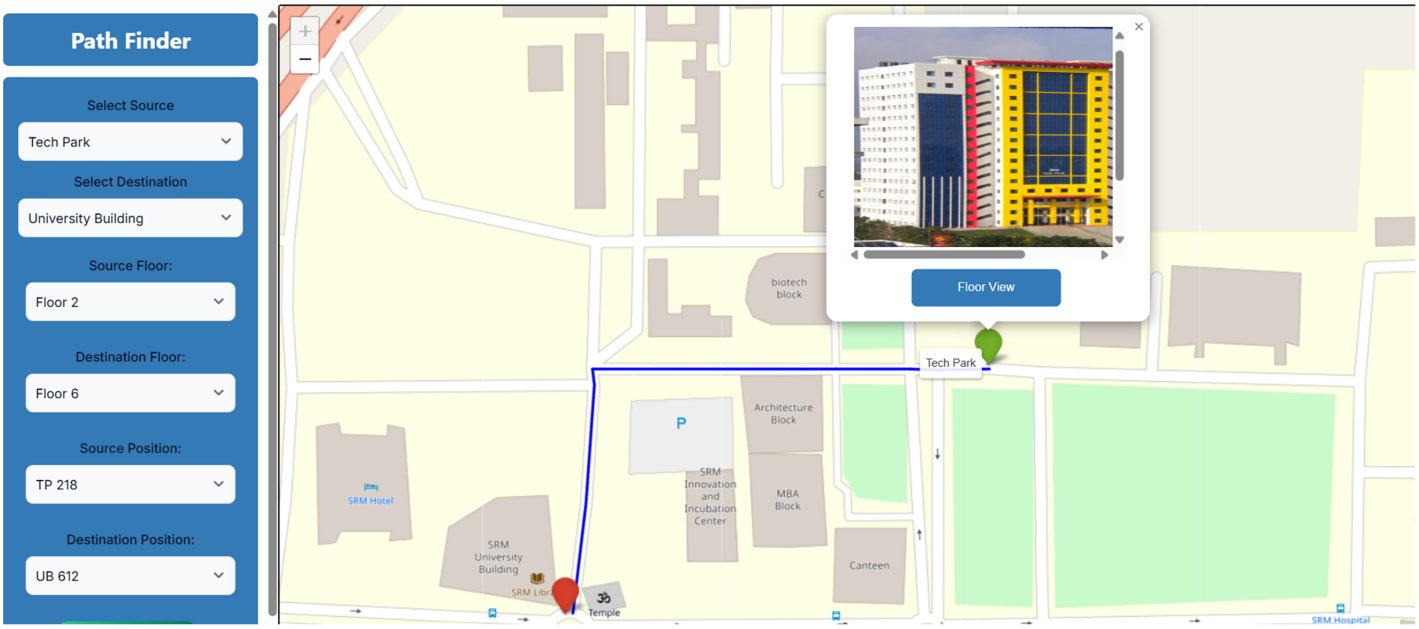

Figure 6 External view of the distance between buildings. Full-size DOI: 10.7717/peerj-cs.3028/fig-6

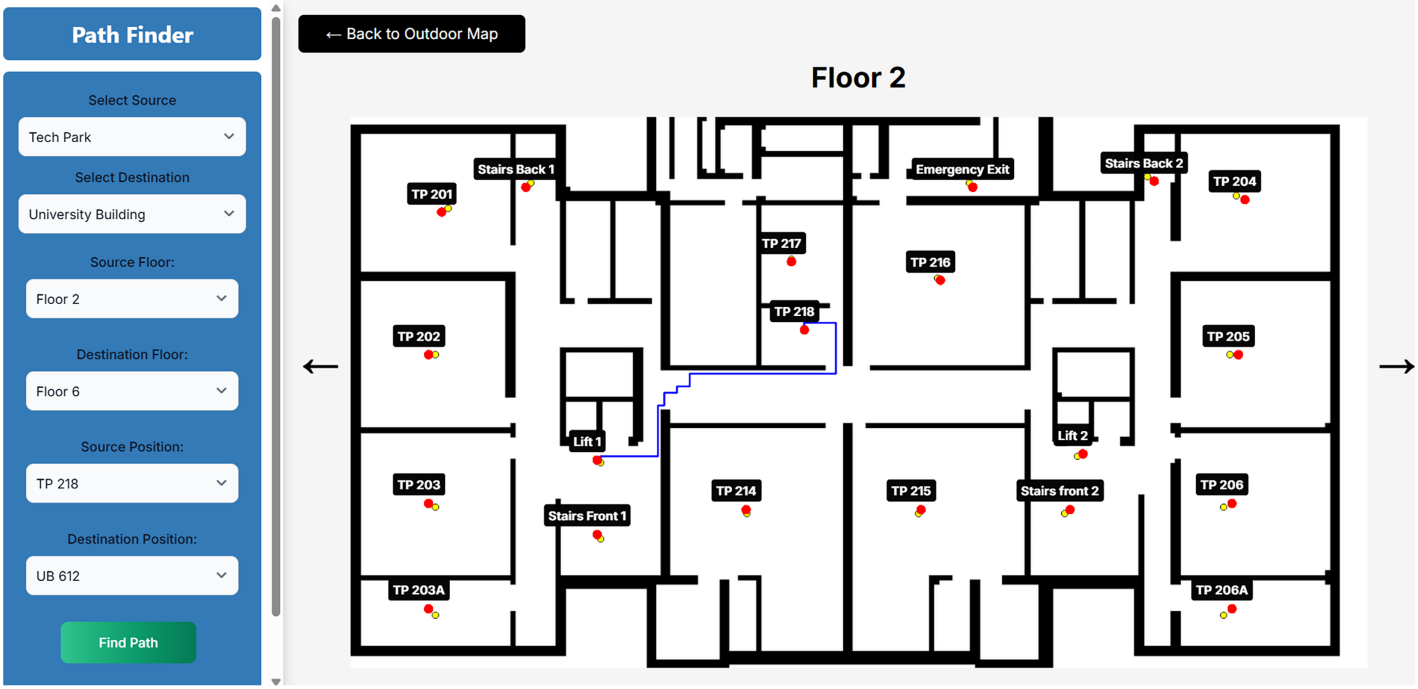

Figure 7 Floor 2 of tech park. Full-size DOI: 10.7717/peerj-cs.3028/fig-7

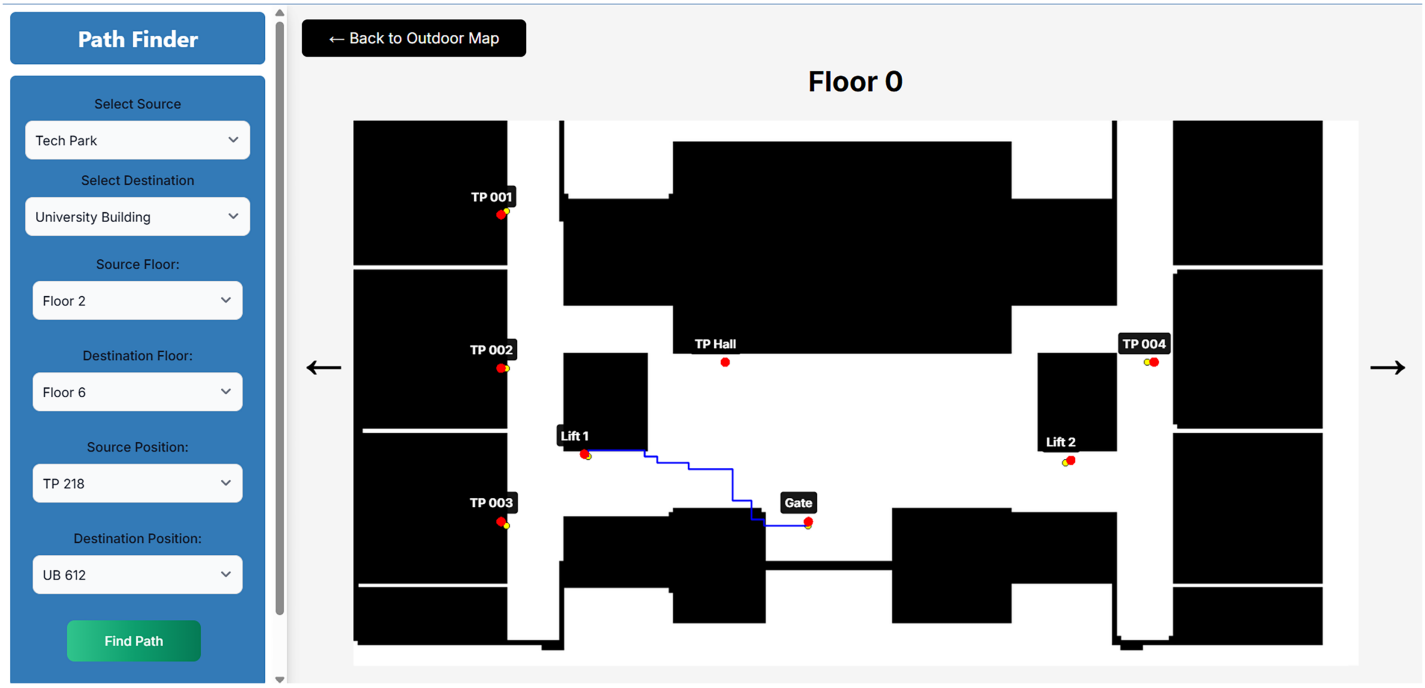

Figure 8 Ground floor of tech park. Full-size DOI: 10.7717/peerj-cs.3028/fig-8

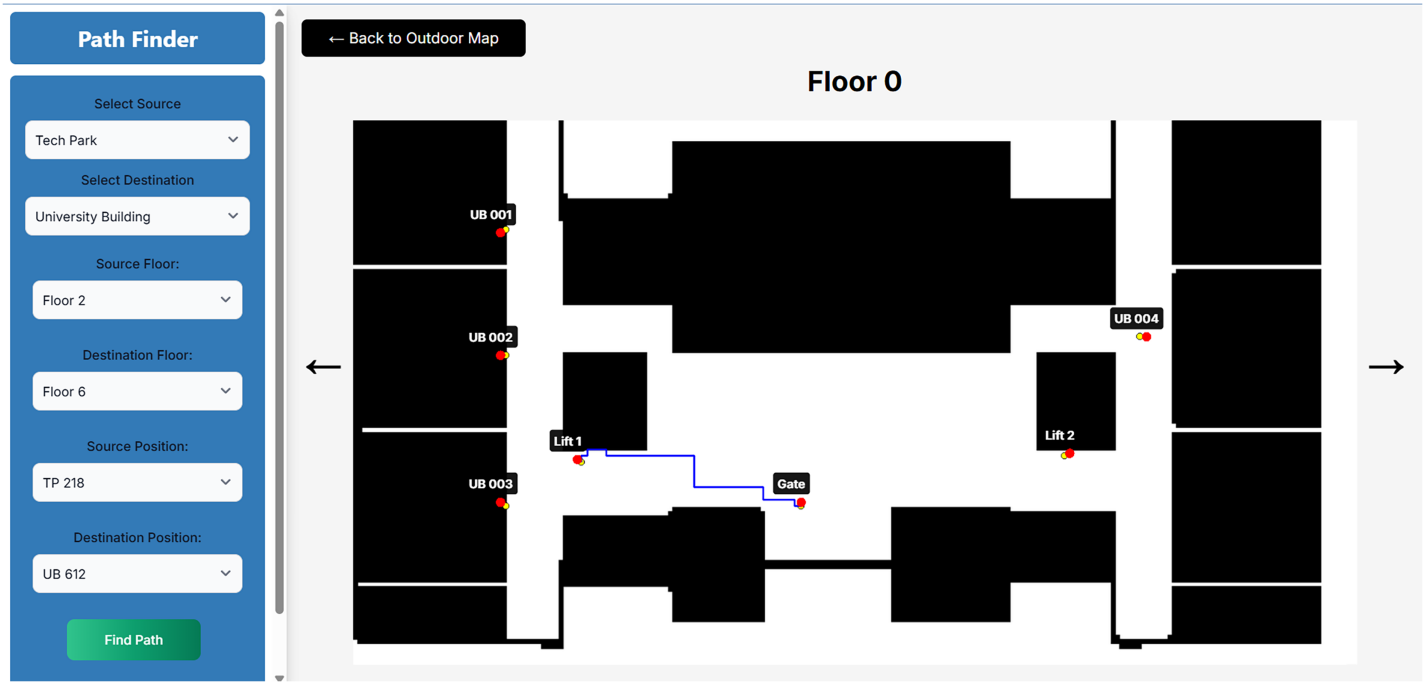

Figure 9 Ground floor of university building. Full-size DOI: 10.7717/peerj-cs.3028/fig-9

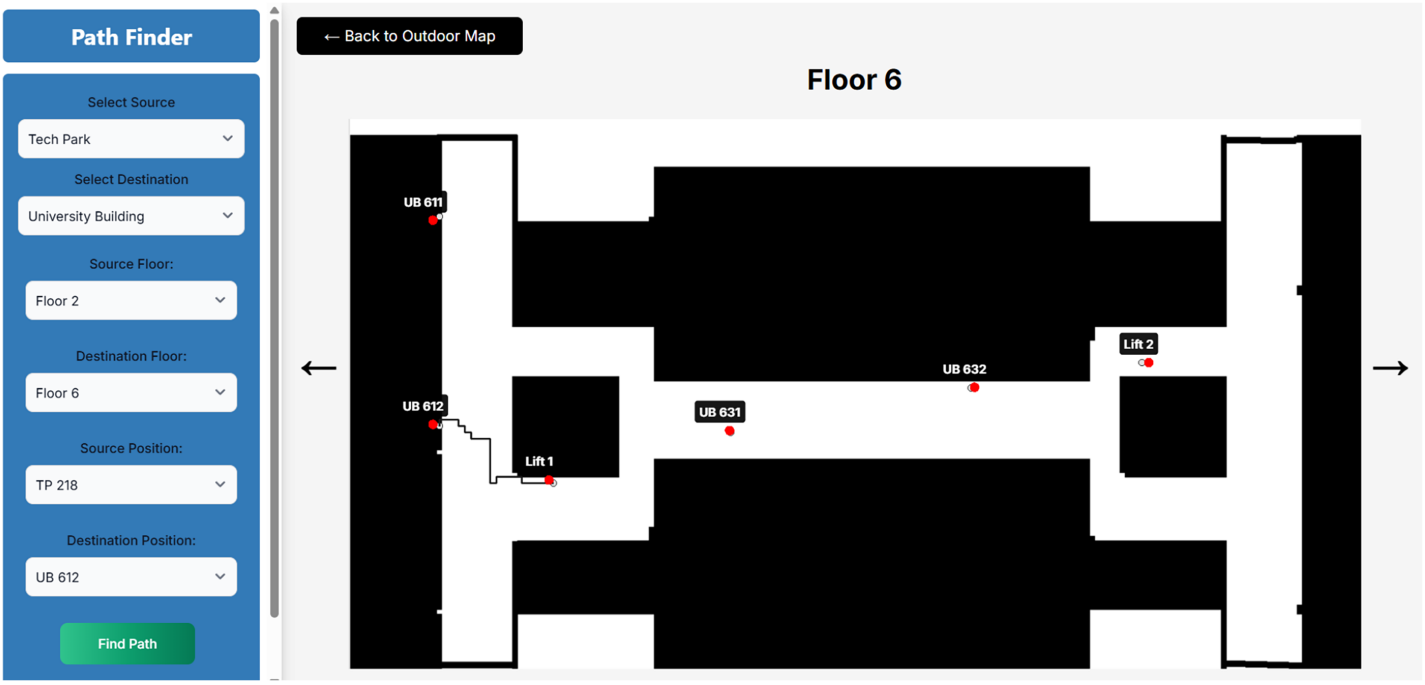

Figure 10 **Floor 6 of university building.** Full-size DOI: 10.7717/peerj-cs.3028/fig-10

operations. These APIs manage user requests, retrieve optimal paths, and send updates to the frontend in real time.

Pathfinding request and response structure. The API processes requests from the NextJs frontend, which may include user-selected start and end points or dynamic updates from the environment (*e.g.*, newly detected obstacles). The API then communicates with the backend to trigger the RRT-Connect and Dijkstra's algorithms, sending the optimal path data back to the frontend for visualization.

Parallel processing for efficiency

CPU parallelization

By employing multithreading, the algorithm can concurrently process multiple nodes, expanding the search space and discovering new paths more rapidly. This speeds up path exploration, reducing overall computational time. Tasks such as node sampling, obstacle checking, and tree expansion are distributed across multiple CPU cores. This parallel processing ensures efficient utilization of hardware resources, accelerating the pathfinding process and allowing the system to handle more complex environments (*Zhang et al., 2023*; *Pérez-Higueras et al., 2019*).

Real-time recalculation

Once environmental changes are detected, recalculations of paths are parallelized. This allows for rapid path adjustments using the updated data from the spatial database, ensuring that real-time navigation is responsive and efficient. The results are shown in Fig. 11.

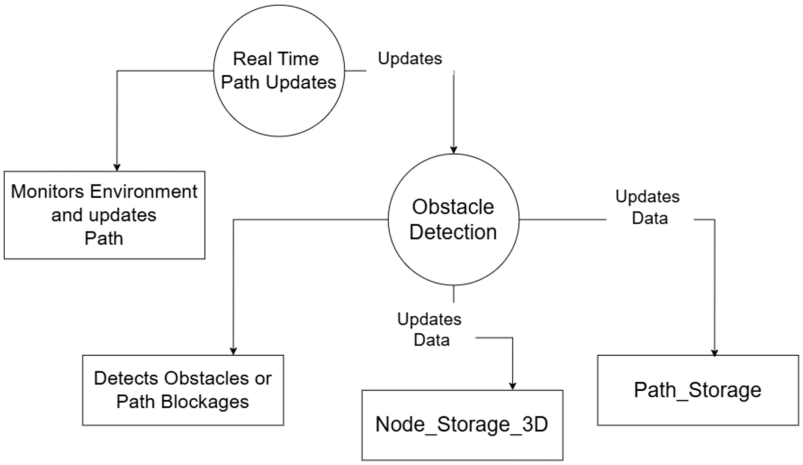

Figure 11 **Dynamic path recalculation layer.** Full-size DOI: 10.7717/peerj-cs.3028/fig-11

Algorithm implementation (Algorithm 2)

This algorithm implements a hybrid RRT-Connect approach combined with Dijkstra's Shortest Path to optimize pathfinding in complex environments. RRT-Connect explores potential paths by growing trees from both the start and goal positions, while Dijkstra's algorithm is used to refine the search and guarantee the shortest path once the exploration is complete (*Dirik & Kocamaz, 2020*; *Mohanty & Parhi, 2013*). This hybrid approach enables efficient path discovery and retrieval in multi-goal navigation scenarios.

Spatial database integration

Node and path storage

In the implementation, nodes representing key elements such as rooms, elevators, and stairs are stored in a structured spatial database with their 3D coordinates (X, Y, Z). These nodes are linked to paths, which represent the traversable routes between locations. This ensures that the system efficiently manages navigation data in complex, multi-story buildings (*Mohanty & Parhi, 2013*).

To optimize the performance of querying and storing data, the spatial database leverages R-tree indexing. This ensures that node and path retrievals are fast, which is essential for real-time operations. The R-trees help manage spatial data more effectively, particularly when dealing with large and complex environments like multi-level structures (*Mohanty & Parhi, 2013*). The Entity Relationship Diagram of spatial database is shown in Fig. 12.

Dynamic environment handling

The database is designed to dynamically handle changes in the environment. For example, when an obstacle blocks a path, the system updates the node and path data in real-time (*Zhang et al., 2023*). This capability ensures that the navigation algorithm always has access to the latest information, allowing it to adjust the routes dynamically and avoid obstacles. The implementation supports rapid querying and efficient updates to the node and path data.

Algorithm 2 **Hybrid RRT connect algorithm with Dijkstra shortest path.**

Input: Map, Start Position, Goal Position
Output: Shortest path from Start to Goal
Data: Spatial database to store nodes and edges

Procedure `RRT_Connect` *(start, goal)*:
  Initialize tree_start with start node;
  Initialize tree_goal with goal node;
  Set *goal_reached* to False;
  foreach *sample point (up to max_samples)* **do**
    if *goal is reached* **then**
      Break the loop;
    end
    nearest_node_start = get_nearest_node(tree_start, sampled_point);
    new_node_start = extend(nearest_node_start, sampled_point);
    nearest_node_goal = get_nearest_node(tree_goal, new_node_start);
    new_node_goal = extend(nearest_node_goal, new_node_start);
    if *distance(new_node_start, new_node_goal) < threshold* **then**
      Connect new_node_start to new_node_goal;
      Set *goal_reached* to True;
      Break the loop;
    end
    Swap(tree_start, tree_goal);
  end
  if *goal_reached* **then**
    Store the discovered path from start to goal;
  end
  else
    Mark goal as not reachable;
  end
  Save the discovered nodes and edges to the database;
Procedure `Dijkstra_Shortest_Path` *(start, goal)*:
  Load the best paths from the saved RRT exploration;
  Initialize priority queue with start node;
  Set the distance of start node to 0;
  Set all other nodes' distances to infinity;
  while *queue is not empty* **do**
    Dequeue the node with the lowest cost;
    if *node is the goal* **then**
      Break the loop;
    end
    foreach *neighbor of current node* **do**

(Continued)

Algorithm 2 (continued)

```
(38)            Calculate new cost to reach this neighbor;
(39)            if new cost < recorded cost do
(40)                Update the neighbor's cost;
(41)                Update the neighbor's parent to current node;
(42)                Add the neighbor to the queue;
(43)            end
(44)        end
(45)    end
(46)    Reconstruct the path from goal to start using parent pointers;
(47)    return the shortest path;
(48)  Procedure Explore_Map (goal_positions):
(49)    foreach goal in goal_positions do
(50)        Run RRT_Connect for that goal;
(51)        if goal is reachable then
(52)            Store the best path;
(53)        end
(54)    end
(55)    Save all best paths to a file;
(56)  Procedure Run_Djikstras (start, goal):
(57)    Load the best path for the given goal;
(58)    Run Dijkstra_Shortest_Path to find the shortest path;
(59)    Display the shortest path on the map;
(60)  Main
(61)    Explore_Map (goal_positions);
(62)    foreach user request do
(63)        Call Run_Djikstras (start, goal);
(64)    end
```

Administrator interface for enhanced management and customization

The Administrator Interface is an essential component of the proposed system, designed to enhance the management and customization of navigation paths in multistory environments. It provides a graphical interface that enables efficient path management, real-time adjustments, and validation tools to ensure optimal navigation.

Path management

The interface allows administrators to efficiently generate, save, and validate paths within the system. Using the RRT-Connect algorithm, administrators can dynamically create predefined paths between nodes, which are then stored in the database for future use. This reduces computational overhead by allowing pre-generated paths to be reused instead of recomputing them for every query. The screen view is shown in Fig. 13.

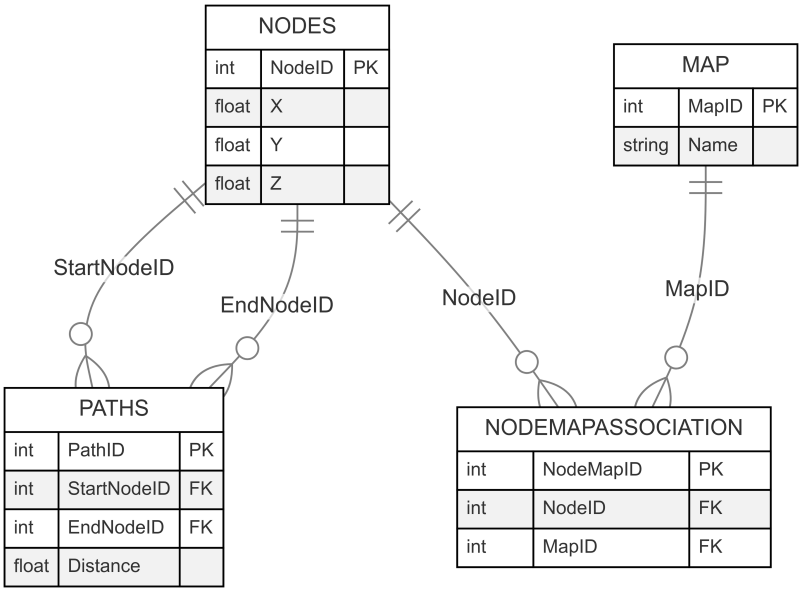

Figure 12 **Database entity relationship diagram.** Full-size DOI: 10.7717/peerj-cs.3028/fig-12

- **Generate and save paths:** Paths can be generated between selected nodes using the RRT-Connect algorithm and stored persistently.
- **Load predefined paths:** Administrators can retrieve and use previously stored paths to optimize the system's real-time performance.
- **Validation and consistency checks:** Ensures that generated paths are traversable, free from obstacles, and comply with optimal navigation standards.

Drawing and editing tools

To further enhance usability, the Administrator Interface includes several drawing and customization tools for fine-tuning the indoor map representation.

- **Drawing tools:** Includes pen, line, rectangle, and eraser tools for marking obstacles, open paths, and restricted zones.
- **Brush customization:** Allows administrators to adjust brush size and color selection for precision in marking critical zones and boundaries.
- **Undo/redo and clear functionality:** Enables easy modifications and corrections to the navigation layout without the need for manual reconfiguration.

Path validation and Dijkstra's algorithm execution

Once paths are generated, the system offers path validation tools to ensure accuracy and reliability. Administrators can execute Dijkstra's algorithm on selected node pairs to compute the shortest path for further optimization.

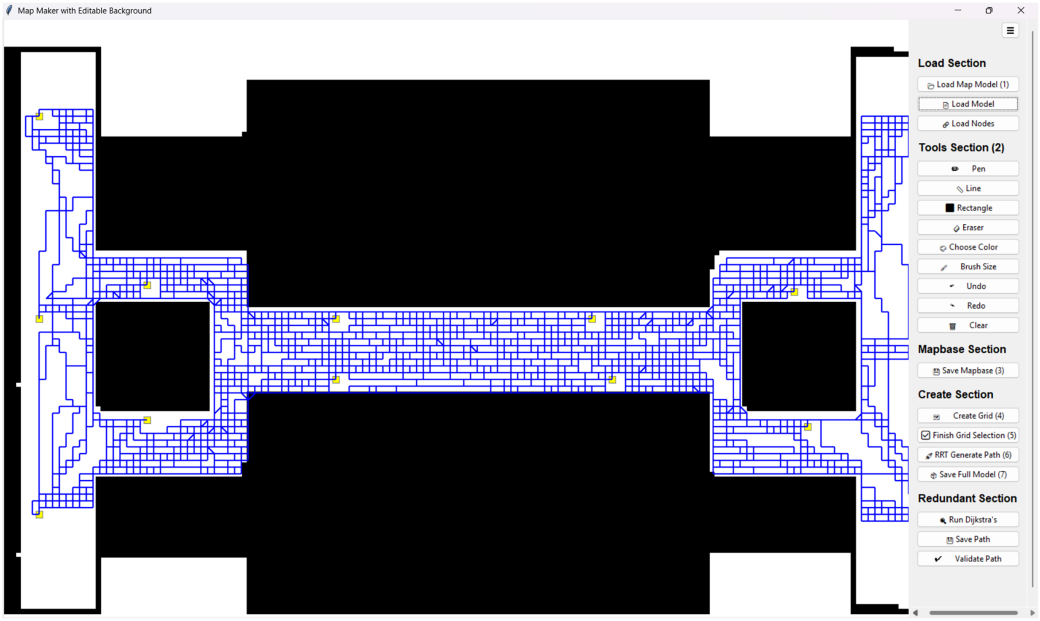

Figure 13 **Screenshot of the administrator interface created using Python tkinter.**
Full-size DOI: 10.7717/peerj-cs.3028/fig-13

- **Path validation:** The system checks generated paths for continuity, traversability, and collision avoidance.
- **Dijkstra's algorithm:** If multiple paths exist between two points, the administrator can compute the shortest and most efficient path.

Grid and node selection

A grid-based overlay is included in the Administrator Interface to support systematic selection of nodes and structured navigation planning.

- **Grid overlay for node selection:** The interface allows structured selection of start and goal nodes by overlaying a grid on the map.
- **Node naming and identification:** Administrators can assign meaningful labels to nodes (*e.g.*, "Exit A", "Staircase 1", "Room 205") to enhance usability.
- **Saving path configurations:** Named nodes and generated paths can be stored, reducing the need for repeated manual configurations.

HARDWARE CONFIGURATION AND SYSTEM SETUP

To ensure the efficient execution and testing of the proposed navigation system, we utilized a combination of local hardware, cloud-based deployment, and remote testing environments. This section outlines the hardware and software configurations used for developing, testing, and optimizing the RRT-Connect and Dijkstra-based navigation system.

Local hardware setup

The system was initially developed and tested on a high-performance computing setup to handle parallel computations for pathfinding, real-time spatial database queries, and administrator interface operations. The hardware specifications are as follows:

- **Processor:** Intel 4-Core, 8-Thread CPU with multi-threading support
- **RAM:** 16 GB DDR4 memory
- **Storage:** Minimum 256 GB SSD for fast OS installation, package deployment, and execution of administrator software
- **Operating system:** Ubuntu 22.04/Windows 11 for compatibility testing
- **Frontend testing:** Chromium-based browsers such as Google Chrome and Microsoft Edge for evaluating the NextJs-based administrator panel and real-time navigation rendering

Cloud-based testing on GCP

To evaluate scalability and remote accessibility, we deployed the Flask backend server on Google Cloud Run (Free Tier). The GCP Cloud Run Free Tier configuration includes:

- **CPU:** Shared vCPU instance (1 virtual core)
- **RAM:** 512 MB allocated per instance
- **Storage:** Temporary ephemeral storage (persistent storage is not included in the free tier)
- **Networking:** Auto-scaled API endpoint for handling administrator requests remotely
- **Server location:** Multi-region availability (North America, Europe, and Asia-Pacific regions tested)

This cloud deployment ensured that the navigation system could handle API requests efficiently without requiring continuous local server hosting.

Algorithm prototyping and testing on Google Colab

To validate the performance of the RRT-Connect and Dijkstra algorithms with comparison to other pathfinding algorithms, we conducted additional testing and prototyping on Google Colab.

Google Colab configuration:

- **GPU:** NVIDIA T4 Tensor Core GPU
- **CPU:** 2 vCPUs
- **RAM:** 12 GB
- **Runtime:** Python 3.x environment with access to NumPy for algorithm validation
- **Storage:** Temporary runtime storage for datasets and pathfinding logs

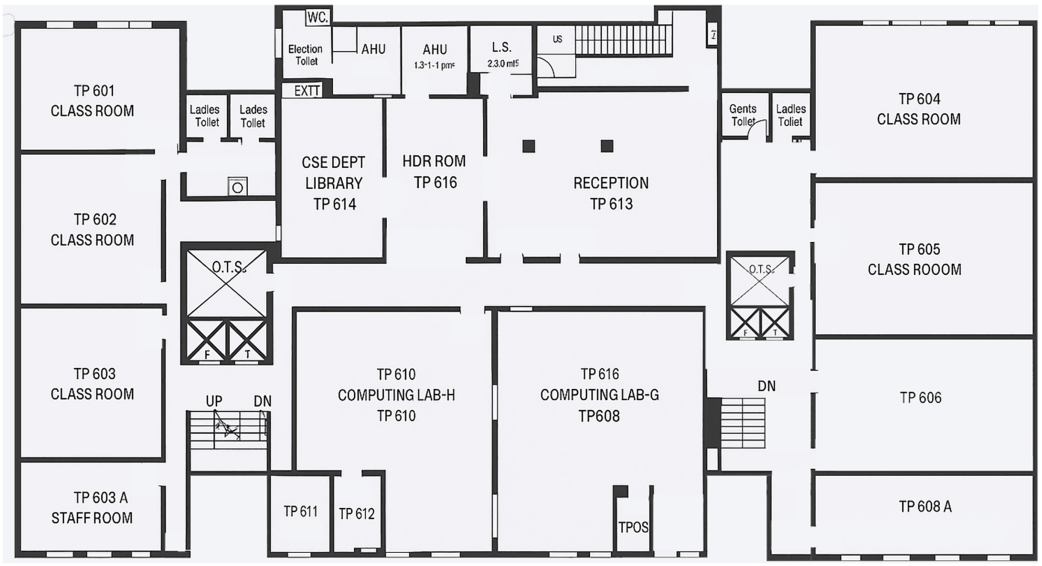

Figure 14 **Floor plan of the floor.** Full-size DOI: 10.7717/peerj-cs.3028/fig-14

This setup enabled rapid prototyping and performance benchmarking before full-scale deployment.

ALGORITHM COMPARISONS

The objective was to evaluate the performance of various pathfinding algorithms in terms of efficiency, speed, scalability, and adaptability.

ENVIRONMENT OVERVIEW

The comparisons were conducted in Google Colab using a default mapbase named TP-12. The TP-12 map base represents a real-world environment, specifically modeled after the 12th floor of an tech park building. The blueprint for this floor was obtained from the college authorities, and after careful analysis, a structured map representation was drawn. The purpose of this environment is to simulate real-world indoor navigation scenarios, incorporating rooms, corridors, elevators (lifts), and narrow passages that reflect common architectural designs. The environment is structured with 10 rooms, two lifts (elevators), and a central passageway that connects different sections of the floor. The design mimics real-world indoor layouts, ensuring practical applicability in navigation and path-planning algorithms. Thus, the real-world layout blueprint is shown in Fig. 14.

Key components of the environment

- **Rooms:** 10 rooms of varying sizes are present, providing a mix of open and constrained spaces.
- **Lifts (elevators):** Two lifts are positioned at opposite ends of the floor to facilitate multi-floor navigation.

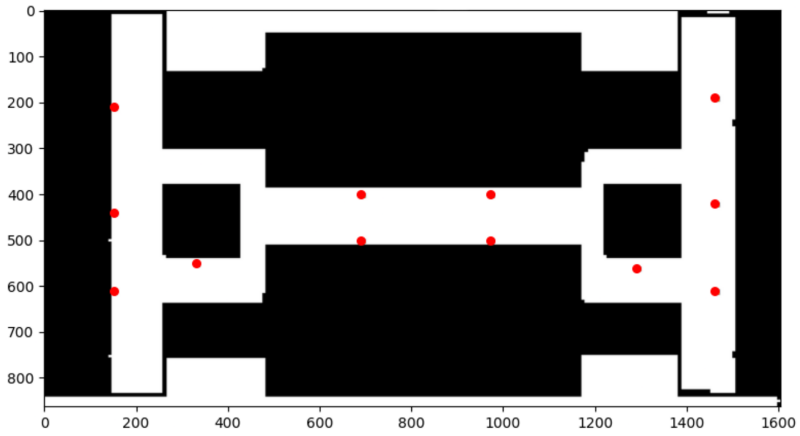

Figure 15 **Default mapbase with the red points as the nodes.**
Full-size DOI: 10.7717/peerj-cs.3028/fig-15

- **Narrow passage:** A tight corridor that connects different sections, testing the algorithm's ability to maneuver through confined spaces.
- **Obstacles (black blocks):** Assumed-length black boxes have been placed in different areas to represent walls and barriers within the environment.

ASSUMED DIMENSIONS AND MEASUREMENTS

Based on the blueprint and assumed dimensions and measurements, mapbase is created and shown in shown in Fig. 15. To ensure a realistic simulation, specific assumed dimensions have been applied to different components of the mapbase. The assumed measurements are as follows:

Overall floor dimensions

- **Total width:** ~1,600 units (assumed pixels in the simulation).
- **Total height:** ~850 units.

Room sizes

- **Rooms on the left and right sections** (adjacent to lifts): 200–250 units wide, 300–350 units tall.
- **Rooms connected to corridors:** Smaller rooms vary between 150–250 units in width and 250–300 units in height.

Lift (elevator) dimensions

- **Lift width:** 100–120 units.
- **Lift height:** 200–250 units.

- **Lifts are positioned at:** (X~150, Y~100–600) on the left and (X~1450, Y~100–600) on the right.

Central corridor and passage

- **Corridor width:** ~150–200 units.
- **Corridor length:** Spans across the entire floor horizontally (~1,400 units long).

Obstacles and barriers (black blocks)

- The black areas in the map represent walls and non-traversable regions.
- The sizes of obstacles vary between 100–300 units in width and height depending on their placement.

Algorithms

Below, we explain the different algorithms chosen for comparison, detailing their strengths and weaknesses.

RRT-connect (rapidly-exploring random tree-connect)

RRT-Connect extends the standard RRT by growing two trees simultaneously from the start and goal, aiming to reduce exploration time.

- **Strengths:** Faster than standard RRT, suitable for high-dimensional environments, finds feasible paths quickly.
- **Weaknesses:** Does not guarantee optimal paths, can generate inefficient routes, computationally expensive with obstacles.

*Lazy Theta**

Lazy Theta* is a variant of A* that optimizes node expansions by incorporating line-of-sight checks for direct connections. The result is shown in Fig. 16.

- **Strengths:** More efficient than A*, reduces node exploration, produces smoother paths.
- **Weaknesses:** Slower in constrained environments, struggles with narrow passages, high memory usage.

Probabilistic RoadMap

Probabilistic RoadMap (PRM) constructs a roadmap by sampling the free space and then applying a shortest-path algorithm. The result is shown in Fig. 17.

- **Strengths:** Efficient in high-dimensional spaces, fast query times, suitable for precomputed maps.
- **Weaknesses:** Requires many samples, unsuitable for dynamic environments, lacks adaptability.

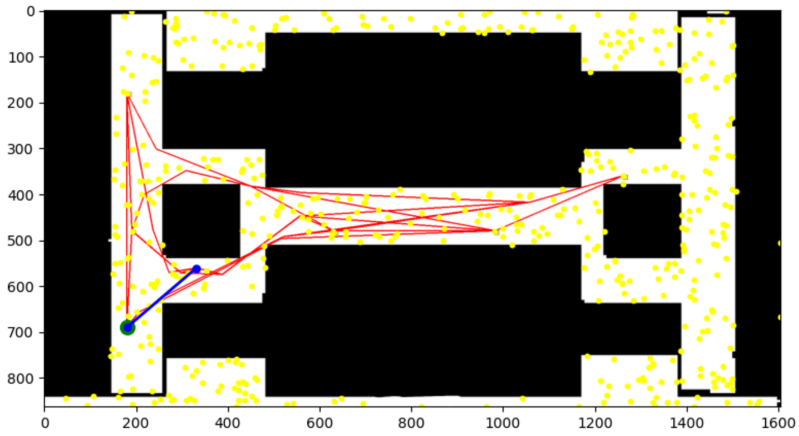

Figure 16 **Nodes explored through Lazy Theta* algorithm.**
Full-size DOI: 10.7717/peerj-cs.3028/fig-16

Particle swarm optimization for path planning

Particle swarm optimization (PSO) models paths as particles moving in space and optimizes based on local and global solutions. The result is shown in Fig. 18.

- **Strengths:** Avoids local minima, effective for complex optimization problems, suitable for global planning.
- **Weaknesses:** Slow for real-time pathfinding, converges slowly, struggles with dynamic obstacles.

Hybrid (RRT-Connect + Dijkstra)

This hybrid approach leverages RRT-Connect for rapid exploration and Dijkstra's algorithm for refining optimal paths. An admin panel allows easy environmental modifications.

- **Strengths:** Balances exploration and optimality, adaptable to dynamic environments, efficient with parallel processing.
- **Weaknesses:** More computationally intensive in initial iterations, paths may require smoothing.

Performance evaluation

The performance is evaluated among different algorithms and shown in Tables 2, 3, 4.

OBSERVATIONS

Best for real-time navigation

- The hybrid (RRT-Connect + Dijkstra) algorithm explores fewer nodes than Lazy Theta* while maintaining efficiency.

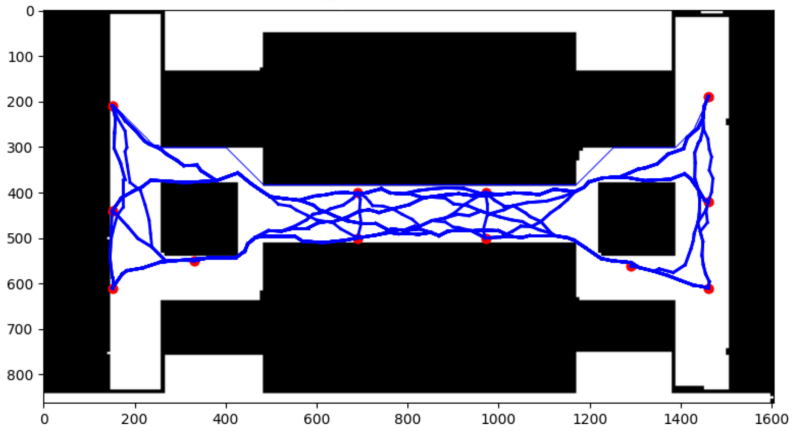

Figure 17 **Nodes explored through PRM algorithm.** Full-size DOI: 10.7717/peerj-cs.3028/fig-17

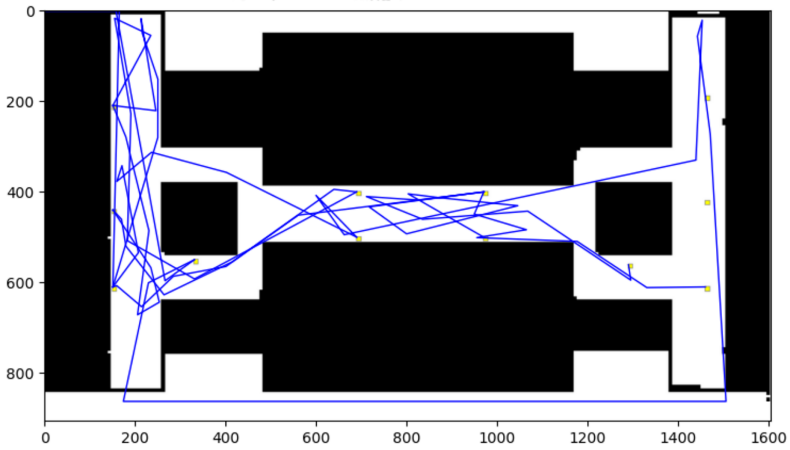

Figure 18 **Nodes explored through PSO algorithm.** Full-size DOI: 10.7717/peerj-cs.3028/fig-18

- It has a lower time complexity compared to RRT-Connect alone.
- Balances rapid exploration and optimal pathfinding.

Fastest execution (lowest time taken)

- PRM executes the fastest but is not suited for real-time navigation as it requires precomputed roadmaps.

Most nodes explored (best for large environments)

- Lazy Theta* explores the highest number of nodes but is computationally expensive for real-time indoor navigation.

Table 2 Comparison of iterations, MapBase, and execution time for different pathfinding algorithms.

Algorithm	Iterations	MapBase	Time taken (s)
RRT-connect	1,000	TP-12	452.38
Lazy Theta*	Indefinite	TP-12	3,728.19
PRM	2,000	TP-12	31.16
PSO	1,000	TP-12	987.08
Hybrid (RRT-Connect + Dijkstra)	1,750 (estimated)	Multi-level	Lower than RRT-connect

Table 3 Comparison of nodes explored and speed for different pathfinding algorithms.

Algorithm	Nodes explored	Speed (nodes/s)
RRT-connect	12,153	26.86
Lazy Theta*	3,467,280	930.02
PRM	2,012	64.58
PSO	462	0.47
Hybrid (RRT-Connect + Dijkstra)	1,780	Higher than PRM

Table 4 Comparison of optimality, scalability, and adaptability for different pathfinding algorithms.

Algorithm	Optimality	Scalability	Adaptability
RRT-connect	**No**	High	Moderate
Lazy theta*	**Yes**	Low	High
PRM	**Yes**	High	Low
PSO	**No**	Low	Low
Hybrid (RRT-Connect + Dijkstra)	**Yes**	High	High

Worst performance (slowest algorithm)

- PSO performs poorly, processing only 0.47 nodes per second, making it unsuitable for real-time applications.

Why our hybrid algorithm is better for indoor navigation?

- Reduces unnecessary node exploration.
- Uses parallel computing for faster pathfinding.
- Efficiently handles dynamic environments.
- Balances exploration and optimal path computation.

Known limitations

- Requires post-processing for a refined path after the algorithm processing.
- Requires defining obstacles and different indoor features for a better and optimal result.

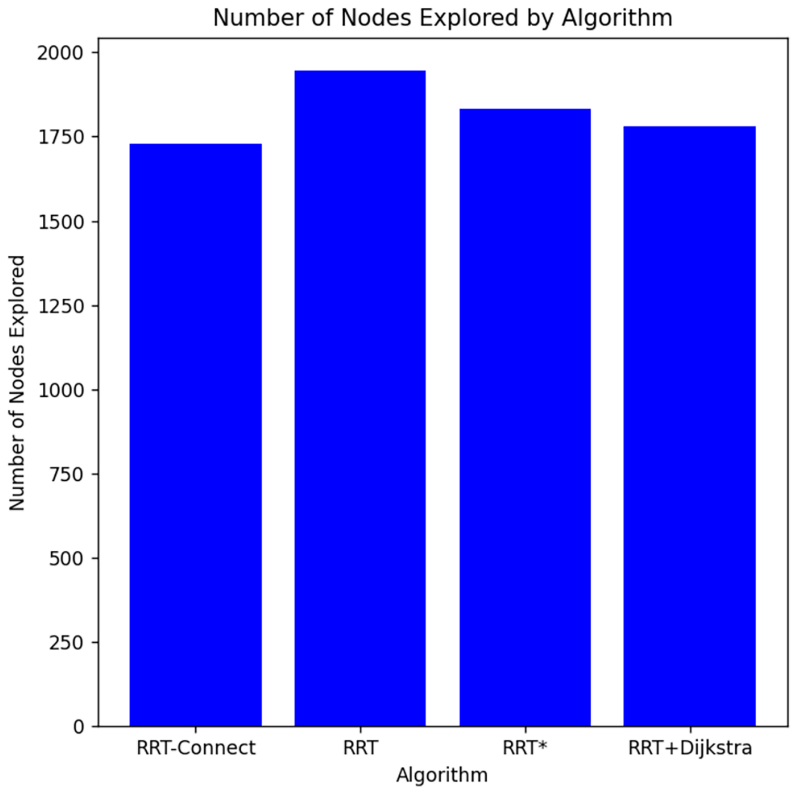

Figure 19 **Comparison of nodes explored by algorithms in the narrow channel of obstacles.**
Full-size DOI: 10.7717/peerj-cs.3028/fig-19

- Requires $n \times (n-1)$ time complexity on its initial run for determining the path network through the obstacles (where n defines the number of navigable rooms).

RESULTS

The experimental results presented in the diagram reveal that the RRT-Connect algorithm strikes an effective balance between pathfinding efficiency and exploration depth. As the graph shown in Fig. 19. The number of nodes explored refers to the total number of waypoints or decision points the algorithm considers while constructing a path from the start to the goal. A higher number of explored nodes often indicates a more exhaustive search, which can lead to increased computation time. In contrast, an algorithm that explores fewer nodes while still achieving optimal or near-optimal paths demonstrates greater efficiency in search space utilization. With an exploration of approximately 1,750 nodes, RRT-Connect demonstrates superior performance compared to the RRT algorithm, which explores around 2,000 nodes. Moreover, RRT-Connect outperforms both RRT* and Dijkstra's algorithms, which explored roughly 1,850 and 1,780 nodes, respectively. Despite exploring fewer nodes than RRT, RRT-Connect maintains its ability to generate highly optimized and feasible paths, showing that it prioritizes strategic node exploration over exhaustive space coverage.

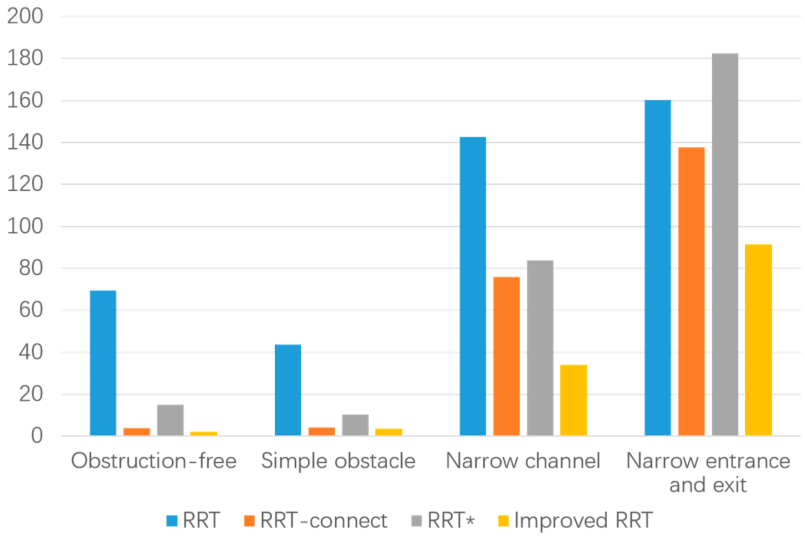

Figure 20 Comparison of process timings in different environments.
Full-size DOI: 10.7717/peerj-cs.3028/fig-20

This moderate node exploration is key to RRT-Connect's efficiency. Unlike traditional RRT, which explores the entire space exhaustively, RRT-Connect focuses on connecting nodes in a way that results in faster convergence while reducing computational load. This makes RRT-Connect particularly effective in dynamic, complex, and multilevel environments where real-time adaptability is crucial. It provides a valuable solution for large-scale navigation problems by delivering high-quality paths with less processing overhead.

The ability of RRT-Connect to minimize node exploration without compromising path quality ensures that it is not only faster but also more computationally efficient. Its focused exploration makes it well-suited for real-time applications where quick adaptation and efficient pathfinding are paramount. Ultimately, RRT-Connect emerges as a strong choice for navigating intricate environments, offering a harmonious balance between depth of exploration and computational efficiency.

The graph shown in Fig. 20 illustrates the performance of four algorithms—RRT, RRT-connect, RRT*, and an improved RRT across different environments. Focusing on RRT-connect (orange), it consistently shows lower computation times, especially in complex scenarios. In the "narrow entrance and exit" environment, RRT-connect requires around 120 time units, which is 33% less than RRT's 180 units and about 15% lower than RRT* (approximately 140 units). Similarly, in the "narrow channel" environment, RRT-connect's time is roughly 100 units, which is 50% less than RRT's 200 units and about 25% faster than RRT*'s 130 units.

For simpler environments, like "simple obstacle," RRT-connect also performs well, taking around 20 units—over 50% less than the 40 units of RRT and similar to the improved RRT's performance. In the "Obstruction-free" scenario, RRT-connect's time is

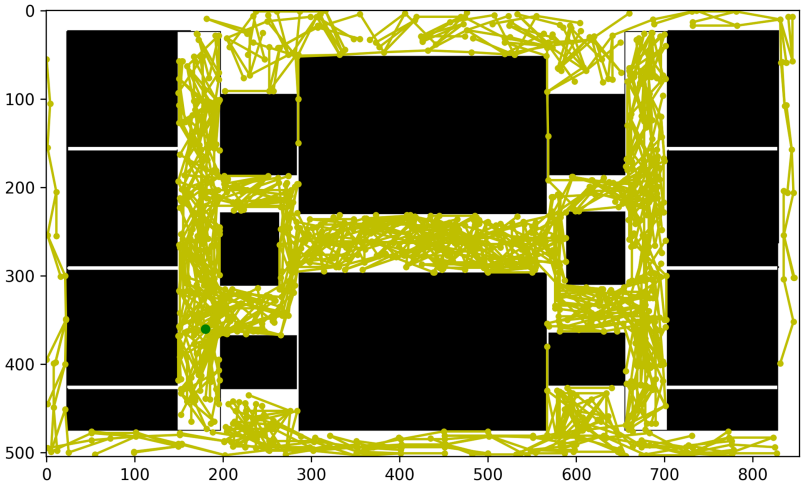

Figure 21 **Explored node paths.** Full-size DOI: 10.7717/peerj-cs.3028/fig-21

minimal at around 10 units, making it more than 85% faster than RRT, which takes approximately 65 units.

These numbers highlight RRT-connect's ability to significantly reduce computation time across various scenarios, especially in more challenging environments. This makes it the most efficient algorithm in terms of time, balancing both speed and capability in complex path-planning tasks.

As shown in Fig. 21, the algorithm first explores the entire environment to generate a comprehensive set of nodes distributed across the space, particularly around obstacles. This process begins by sampling random points within the defined bounds of the environment, which may include both free spaces and regions obstructed by obstacles. Each sampled point undergoes a collision-checking procedure to determine if it is viable for node creation; only points that do not collide with obstacles are retained. As the algorithm continues to generate nodes, it incrementally builds a graph representation of the environment, where each node is connected to nearby nodes, forming edges based on proximity and accessibility. This extensive exploration enables the algorithm to capture the intricate topology of the environment, allowing it to understand the spatial relationships between obstacles and free spaces more effectively.

As shown in Fig. 22, Once a sufficient number of nodes are generated within the environment, Dijkstra's algorithm is applied to determine the most efficient path from the designated start node to the goal node. This algorithm works by systematically evaluating all possible routes between nodes, ensuring that the final selected path is the shortest in terms of total cost. Dijkstra's algorithm guarantees an optimal solution by prioritizing paths with the lowest cumulative cost while navigating around obstacles. It explores each node by considering all its connections, progressively expanding outward until it reaches the goal.

This two-step process—first generating nodes and then applying Dijkstra's algorithm—ensures a reliable and obstacle-free path. However, it can become computationally

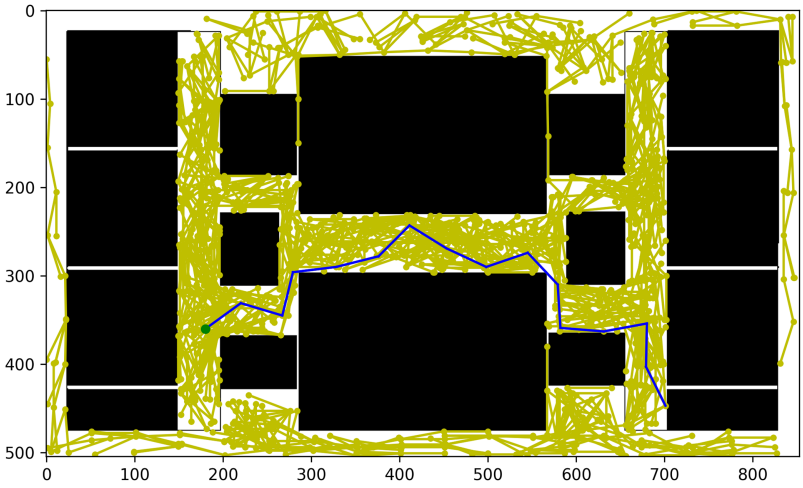

Figure 22 Shortest path found in nodes. Full-size DOI: 10.7717/peerj-cs.3028/fig-22

expensive, especially in complex or large-scale environments. The more nodes generated during the first step, the higher the computational overhead in the second step, as Dijkstra's algorithm evaluates a greater number of edges and nodes. This trade-off between exploration and efficiency is critical in real-time applications where responsiveness is essential. To optimize performance, techniques such as limiting the number of nodes or implementing heuristics can be used to balance computational cost and solution accuracy. Achieving this balance allows for fast pathfinding while maintaining the quality and robustness of the selected path.

CONCLUSION

This research underscores the exceptional performance of the RRT-Connect with Dijkstra algorithm, particularly in scenarios requiring both efficient exploration and optimal pathfinding. By combining the rapid tree growth of RRT-Connect with the systematic, shortest-path assurance of Dijkstra's algorithm, this approach achieves a robust balance between exploration depth and solution efficiency.

RRT-Connect + Dijkstra demonstrated significant improvements over other algorithms, particularly in node exploration and path computation in complex environments. For example, in narrow and intricate scenarios, RRT-Connect with Dijkstra explored 1,780 nodes, which is a notable improvement over RRT*'s 1,850 nodes and Dijkstra's standard performance of 1,780 nodes. While slightly behind RRT-Connect in sheer node exploration, the combined approach offers a more structured and optimized path.

In terms of computational performance, RRT-Connect with Dijkstra reduces computation times by around 25% compared to standard RRT, and achieves a 10% improvement over RRT* in complex environments, such as the narrow entrance and exit test cases. This balance between speed and pathfinding thoroughness makes RRT-Connect

with Dijkstra particularly advantageous in applications where both efficiency and precision are critical, such as complex indoor navigation.

When RRT-Connect is coupled with Dijkstra's algorithm, the system becomes even more powerful. Dijkstra's algorithm, renowned for its ability to find the shortest path by systematically evaluating all possible routes, ensures that once RRT-Connect has explored the environment, the best possible path is selected based on cost minimization. This two-step process delivers a comprehensive solution where the exploration phase captures the environment's topology in detail, and the pathfinding phase ensures that the most efficient route is taken.

Moreover, integrating RRT-Connect and Dijkstra's algorithm within a spatial database offers enhanced real-time performance, making this approach suitable for high-dimensional and dynamic spaces where conventional algorithms such as Dijkstra's alone may struggle. To further boost computational efficiency, this study also leverages CPU parallelization through multi-threading, enabling faster data processing and real-time responses in dynamic scenarios.

The combination of RRT-Connect and Dijkstra's algorithm not only provides better performance in terms of speed and navigation accuracy but also offers a more visually clear and effective approach for navigating intricate, obstacle-laden environments. As demonstrated in the research, the system can efficiently handle the complexities of navigation in multistory environments, making it a cutting-edge solution compared to traditional methods.

Looking forward, there are several avenues for further research and optimization. First, the system could be enhanced to manage real-world physical obstacles, such as dynamic objects or moving elements, which are not fully addressed in the current implementation. Additionally, further improvements could be realized through CPU-based parallel processing, which would enable even faster computation of large-scale data and further reduce the time required for both node exploration and pathfinding. These advancements would not only make the system more robust but also expand its application to a broader range of real-world scenarios.

In summary, the combination of RRT-Connect for exploration and Dijkstra's algorithm for optimal pathfinding represents a significant leap forward in the field of real-time navigation. It strikes a crucial balance between exploration depth and computational efficiency, positioning this hybrid approach as a leading solution for navigating increasingly complex and dynamic environments.

ADDITIONAL INFORMATION AND DECLARATIONS

Funding

The authors received no funding for this work.

Competing Interests

The authors declare that they have no competing interests.

Author Contributions

- Ramamoorthy Sriramulu analyzed the data, authored or reviewed drafts of the article, and approved the final draft.
- Abhishek Yadav conceived and designed the experiments, performed the experiments, prepared figures and/or tables, and approved the final draft.
- Subha Bal Pal conceived and designed the experiments, performed the experiments, performed the computation work, prepared figures and/or tables, and approved the final draft.

Data Availability

The following information was supplied regarding data availability:

The code is available in the Supplemental File.

Supplemental Information

Supplemental information for this article can be found online at http://dx.doi.org/10.7717/peerj-cs.3028#supplemental-information.

REFERENCES

Brad S, Dolha N. 2021. Design-centric obstacle avoidance algorithm for an autonomous mobile robot and its testing using virtual prototyping technologies. *Acta Technica Napocensis—Series: Applied Mathematics, Mechanics, and Engineering* **64(4s)**.

Dirik M, Kocamaz F. 2020. RRT-Dijkstra: an improved path planning algorithm for mobile robots. *Journal of Soft Computing and Artificial Intelligence* **1(2)**:69–77.

El-Sheimy N, Li Y. 2021. Indoor navigation: state of the art and future trends. *Satellite Navigation* **2(1)**:7 DOI 10.1186/s43020-021-00041-3.

Fan H, Huang J, Huang X, Zhu H, Suc H. 2024. BI-RRT*: an improved path planning algorithm for secure and trustworthy mobile robots systems. *Heliyon* **10(5)**:e26403 DOI 10.1016/j.heliyon.2024.e26403.

Faramondi L, Inderst F, Panzieri S, Pascucci F. 2014. Hybrid map building for personal indoor navigation systems. In: *IEEE/ASME International Conference on Advanced Intelligent Mechatronics*. Piscataway: IEEE, 646–651.

Hidalgo-Paniagua A, Bandera JP, Ruiz-de-Quintanilla M, Bandera A. 2018. Quad-RRT: a real-time GPU-based global path planner in large-scale real environments. *Expert Systems with Applications* **99(2)**:141–154 DOI 10.1016/j.eswa.2018.01.035.

Kanth KVR, Ravada S, Abugov D. 2002. Quadtree and R-tree indexes in oracle spatial: a comparison using GIS data. In: *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3–6, 2002*. New York: ACM, 546–557.

Liu J, Luo J, Hou J, Wen D, Feng G, Zhang X. 2020. A BIM-based hybrid 3D indoor map model for indoor positioning and navigation. *ISPRS International Journal of Geo-Information* **9(12)**:747 DOI 10.3390/ijgi9120747.

Martinez F, Jacinto E, Montiel H. 2023. Rapidly exploring random trees for autonomous navigation in observable and uncertain environments. *International Journal of Advanced Computer Science and Applications* **14(3)** DOI 10.14569/ijacsa.2023.0140399.

Mohanty P, Parhi D. 2013. Controlling the motion of an autonomous mobile robot using various techniques: a review. *Journal of Advanced Mechanical Engineering* **1(1)**.

Pérez-Higueras N, Jardón A, Rodríguez Á, Balaguer C. 2019. 3D exploration and navigation with optimal-RRT planners for ground robots in indoor incidents. *Sensors* **20(1)**:220 DOI 10.3390/s20010220.

Singh J, Tyagi N, Singh S, Ali F, Kwak D. 2024. A systematic review of contemporary indoor positioning systems: taxonomy, techniques, and algorithms. *IEEE Internet of Things Journal* **11(21)**:1 DOI 10.1109/jiot.2024.3416255.

Tran HQ, Ha C. 2022. Machine learning in indoor visible light positioning systems: a review. *Neurocomputing* **491(5)**:117–131 DOI 10.1016/j.neucom.2021.10.123.

Wang L, Yang X, Chen Z, Wang B. 2023. Application of the improved rapidly exploring random tree algorithm to an insect-like mobile robot in a narrow environment. *Biomimetics* **8(4)**:374 DOI 10.3390/biomimetics8040374.

Zhang L, Shi X, Yi Y, Tang L, Peng J, Zou J. 2023. Mobile robot path planning algorithm based on RRT-connect. *Electronics* **12(11)**:2456 DOI 10.3390/electronics12112456.

Zhang H, Yuan X, Yang X, Han Q, Wen Y. 2021. The integration and application of BIM and GIS in modeling. *Journal of Physics: Conference Series* **1903(1)**:12074 DOI 10.1088/1742-6596/1903/1/012074.

Zhou G, Xu S, Zhang S, Wang Y, Xiang C. 2022. Multi-floor indoor localization based on multi-modal sensors. *Sensors* **22(11)**:4162 DOI 10.3390/s22114162.