Robotics

Minor in AI - IIT ROPAR

June 11, 2025

1 Motion Planning in Robotics

What is Motion Planning?

Motion planning is a core component of robotics that deals with the computation of a valid and often optimal path or trajectory that a robot must follow to reach a goal configuration from a given start configuration. This path must be planned such that the robot does not collide with any obstacle in its environment.

Formally, motion planning can be defined as the problem of determining a sequence of valid configurations, or poses, that move the robot from an initial position to a goal position while satisfying a set of constraints, including:

- Avoidance of static and dynamic obstacles
- Respecting the kinematic and dynamic constraints of the robot
- Optimization of a cost function (e.g., time, distance, energy)

The essence of motion planning can be compared to the fundamental human task of navigation. Every time we try to get from point A to point B, either on foot or in a vehicle, our brain solves a form of the motion planning problem.

Problem Statement of Motion Planning

The motion planning problem can be posed as follows:

- **Objective:** Find an optimal or feasible path for the robot to move from a start pose to a goal pose.
- **Constraints:**
  - The path must avoid collisions with obstacles.
  - The path must be realizable given the robot's geometry and kinematic capabilities.

Given:

- A start configuration (pose) of the robot, usually defined in a coordinate frame.
- A goal configuration that the robot must reach.
- A geometric and possibly dynamic model of the robot (e.g., shape, DOF, joint limits).
- A geometric model of the world or workspace (e.g., location and shape of obstacles).

The output of a motion planner is a continuous path or discrete trajectory that satisfies all constraints and optimizes (or at least satisfies) a given performance metric such as time, distance, energy consumption, or safety.

Real-Life Example: Motion Planning in Everyday Life

Let us consider a familiar, real-life example: using a navigation app to find directions.

- You unlock your smartphone and open a navigation application (like Google Maps).
- You input your current location (say, your house) as the start point.
- You type in a destination, such as "Domino's Pizza Restaurant".

The application then internally models a graph of roads and paths between locations. It computes multiple possible routes using various algorithms such as A*, Dijkstra's Algorithm, or probabilistic roadmaps. The criteria for route selection may include:

- **Shortest distance:** Minimize the number of kilometers/miles.
- **Least time:** Minimize the expected travel time, possibly accounting for traffic.
- **User preferences:** Avoid toll roads, highways, or choose scenic routes.

This example maps directly to robotic motion planning:

- The road network is analogous to the robot's *configuration space* (C-space).
- The robot's body, size, and joint limits correspond to constraints similar to the vehicle's width and maneuverability.
- Obstacles like buildings, potholes, or roadblocks resemble physical obstructions in a robot's workspace.

Case Study: Warehouse Robot Motion Planning

Consider an autonomous robot deployed in a warehouse to transport items from storage shelves to a packing station.

Scenario Description:

- The warehouse floor is a grid-like structure with narrow aisles.
- The robot starts at a charging dock (start pose).
- The packing station is the goal location.
- Several shelves and obstacles are placed in the environment.

Challenges Faced:

- The robot must avoid bumping into shelves, other robots, and humans.
- The motion planner must compute a path that is not only feasible but also efficient — minimizing delivery time.
- If other robots are moving, then the planner must incorporate dynamic obstacles and predict their future motion.

To solve this, the robot may employ a hybrid planning approach:

- **Global planning:** A* or Dijkstra's algorithm over a grid or graph representation of the environment.
- **Local planning:** Dynamic Window Approach (DWA) or Velocity Obstacle methods for reactive obstacle avoidance.
- **Behavioral planning:** Task-level planning to sequence multiple pickups and deliveries efficiently.

Case Study: Self-Driving Cars

Self-driving cars present an advanced and complex case of motion planning.

Environment:

- Urban road networks with intersections, traffic signals, and pedestrians.
- Dynamic obstacles like other cars, cyclists, and erratic behavior of pedestrians.
- Constraints like speed limits, lane boundaries, and real-time sensor data.

The motion planning problem becomes multi-layered:

- **Strategic Planning:** Determine the long-term route to the destination.
- **Tactical Planning:** Decide whether to overtake, yield, or wait.
- **Operational Planning:** Generate fine-grained, real-time trajectories.

Here, motion planning incorporates:

- Continuous replanning due to dynamic changes in the environment.
- Use of machine learning for behavior prediction.
- Fusion of data from LIDAR, RADAR, GPS, and cameras.

Types of Motion Planning Approaches

- **Graph-based Planners:**
  - Examples: A*, Dijkstra's algorithm.
  - Suitable for known, static environments.
  - Require discretization of space into grids or graphs.
- **Sampling-based Planners:**
  - Examples: PRM (Probabilistic Roadmap), RRT (Rapidly-exploring Random Tree).
  - Good for high-dimensional spaces.
  - Do not require an explicit representation of free space.
- **Optimization-based Planners:**
  - Examples: CHOMP, TrajOpt.
  - Use cost functions to find smooth, feasible trajectories.
  - Often used in combination with sampling-based planners.

2 Applications of Motion Planning in Robotics

Overview

Motion planning is not confined to a single domain of robotics. It is an essential functionality that finds application in virtually every robotic system where purposeful motion is required. From autonomous ground vehicles navigating complex road networks to robotic arms delicately assembling mechanical components, motion planning underpins the robot's ability to operate effectively in structured and unstructured environments.

At its core, motion planning enables a robot to decide *how to move*. This involves identifying a safe, efficient, and feasible path that satisfies various physical, environmental, and task-specific constraints. Applications of motion planning are vast, and they span across fields as diverse as industrial automation, autonomous driving, aerial robotics, warehouse logistics, healthcare robotics, and even space exploration.

1. Mobile Robotics

One of the most prominent and intuitive applications of motion planning lies in mobile robotics, where robots are required to navigate in 2D or 3D environments.

Use Cases:

- **Autonomous Ground Robots:** Wheeled robots in indoor or outdoor environments must compute collision-free paths from their current location to a desired goal.
- **Aerial Drones:** Flying robots must plan paths in 3D space, avoiding buildings, trees, and even other drones in motion.
- **Underwater Vehicles:** Similar planning is required for robots operating beneath the surface of oceans, where GPS may not work and obstacles may be dynamic and uncertain.

Case Study: Campus Delivery Robot

Imagine a delivery robot on a university campus designed to transport food or books between departments. The robot must:

- Plan routes through open corridors, hallways, and outdoor paths.
- Avoid static obstacles such as benches, trees, or staircases.
- Dynamically replan in response to pedestrians, construction zones, or weather changes.

The motion planning algorithm used may involve:

- Global planners (e.g., A*, Dijkstra) to compute a long-range plan.
- Local planners (e.g., DWA - Dynamic Window Approach) to navigate in real time.

2. Industrial Manipulators

In industrial settings, robotic manipulators or arms are used for tasks such as welding, assembly, painting, or pick-and-place operations. Motion planning in this context focuses on finding feasible joint-space or Cartesian-space trajectories.

Challenges:

- **Kinematic Constraints:** The robot's joints may have limits on angles, speeds, or accelerations.
- **Obstacle Avoidance:** The robot must avoid hitting itself (self-collision) or its surroundings (environmental collision).
- **Precision:** Movements must be highly accurate, especially in tasks like micro-assembly or surgical automation.

Case Study: Automotive Assembly Line

In a car manufacturing plant, robotic arms are employed to weld car doors onto the chassis.

- The work area is cluttered with other robots and large components.
- The robot must follow a precise sequence of operations.
- Paths must be smooth, collision-free, and optimized for cycle time.

Motion planners such as RRT-Connect, CHOMP, or TrajOpt may be used to compute smooth, feasible joint trajectories in constrained spaces.

3. Self-Driving Cars

Perhaps one of the most challenging and richly layered domains for motion planning is autonomous driving. A self-driving car must make real-time decisions to safely and efficiently navigate urban and highway environments.

Types of Planning Layers:

- **Route Planning (Global):** Plan from city A to city B using a map.
- **Behavior Planning (Tactical):** Decide whether to change lanes, overtake, or stop.
- **Trajectory Planning (Operational):** Compute a precise trajectory that the vehicle's actuators can follow.

Case Study: Urban Navigation During Rush Hour

A self-driving car is navigating downtown traffic at 6 PM:

- Dynamic obstacles (pedestrians, cyclists, other cars) are everywhere.
- Traffic rules and signals must be obeyed.
- Sudden changes (e.g., jaywalking, merging vehicles) require immediate re-planning.

The motion planner must ensure:

- Safety (no collisions)
- Comfort (smooth acceleration/deceleration)
- Efficiency (avoiding unnecessary stops)

Here, planners like Hybrid A*, MPC (Model Predictive Control), or lattice-based planners are often used.

4. Other Significant Applications

Beyond conventional mobile and industrial robots, motion planning is critical in many specialized applications:

Warehouse Robots

- Robots such as Amazon's Kiva system must efficiently retrieve and transport items.
- Multiple robots coordinate motion to avoid collisions and minimize idle time.
- Motion planners ensure warehouse layouts are fully utilized without congestion.

Surgical Robotics

- Minimally invasive surgical robots must plan tool paths within constrained areas of the human body.
- Safety and precision are paramount.
- Real-time planning is often needed in response to tissue deformation or patient movement.

Space Robotics

- Rovers on planetary surfaces (like Mars) use motion planning to navigate rough, unknown terrain.
- Robotic arms on space stations plan collision-free paths to perform repairs or satellite servicing.

Agricultural Robotics

- Robots must navigate uneven fields while avoiding crops and obstacles.
- Motion planning helps with harvesting, seeding, or spraying in dynamic environments.

Humanoid Robots

- Bipedal robots require motion planning not only for path navigation but also for balance and gait planning.
- Motion planning becomes multi-objective: maintaining stability, optimizing steps, and adapting to terrain.

3 Key Concepts in Robot Movement Planning

The process of enabling robotic systems to move meaningfully in their environments can be broken down into three foundational layers:

1. **Path Planning**
2. **Trajectory Generation**
3. **Motion Planning**

These three concepts build on one another, evolving from abstract geometric planning to detailed, executable movement. Each layer introduces additional considerations—starting from simple configuration sequences to physically constrained, dynamic behavior.

1. Path Planning

Path planning is the first and most fundamental step in robot motion. It deals with the purely geometric problem of finding a collision-free path from a start configuration to a goal configuration within the environment. This stage does not account for how fast the robot moves or the physical limitations of the robot's actuators.

Definition and Focus

- **Focus:** Compute a feasible, collision-free path that avoids obstacles and connects the start and goal configurations.
- **Output:** A sequence of configurations (positions or joint angles) that form a continuous, obstacle-free path in configuration space (C-space).
- **Nature:** Geometric and abstract; ignores timing, dynamics, and velocities.

Example: GPS Navigation

Consider a GPS navigation system that suggests a route from your house to a restaurant:

- It computes possible routes using road networks and map data.
- It avoids roadblocks or areas under construction.
- It finds the shortest or fastest route based on distance or estimated time.

While the output is a route or sequence of streets, the system does not instruct you how fast to drive on each segment — that would be the next layer (trajectory generation).

Scope and Techniques

- **Scope:** Purely geometric; lays the groundwork for further processing.
- **Common Techniques:**
  - Graph Search: A*, Dijkstra
  - Sampling-Based: PRM, RRT
  - Visibility Graphs, Voronoi Diagrams

2. Trajectory Generation

Once a geometric path has been computed, the next task is to generate a trajectory — a time-parameterized plan that tells the robot where to be at every instant. Trajectory generation involves assigning time, velocity, and acceleration to each segment of the path while considering dynamic feasibility and physical constraints of the robot.

Definition and Focus

- **Focus:** Convert a geometric path into a smooth, time-aware trajectory that can be physically executed.
- **Output:** A function mapping time to configuration: $\mathbf{q}(t), \dot{\mathbf{q}}(t), \ddot{\mathbf{q}}(t)$.
- **Goal:** Ensure smoothness, avoid sudden accelerations, and stay within mechanical and actuation limits.

Example: Robotic Arm Performing Pick and Place

Imagine a robotic arm moving between two locations on an assembly line:

- A path planner has given it a collision-free path.
- However, the movement must be smooth — no jerky motion.
- The arm must accelerate gently, maintain constant speed in the middle, and decelerate before stopping.

Trajectory generation applies time-based profiles to this path, ensuring safe and efficient operation.

Key Characteristics and Methods

- **Smoothness:** Ensures that velocity and acceleration changes are continuous.
- **Feasibility:** Respects robot's velocity, acceleration, and jerk limits.
- **Physical Constraints:** Takes inertia and actuator capabilities into account.
- **Common Methods:**
  - Polynomial Trajectories (e.g., cubic, quintic splines)
  - Spline Interpolation (B-splines, Bézier curves)
  - Velocity Profiling (trapezoidal, S-curve)
  - Time-scaling techniques

Scope

Trajectory generation acts as the bridge between abstract planning and real-world execution. It transforms static waypoints into executable movement commands by defining the motion's temporal profile.

3. Motion Planning

Motion planning is the full-stack approach to robotic movement. It encompasses both path planning and trajectory generation but further integrates robot kinematics, dynamics, feedback control, real-time sensor data, and stability constraints. This stage ensures that the robot can move safely and effectively in a real-world setting, adapting to uncertainty and disturbances.

Definition and Focus

- **Focus:** Integrate geometry, timing, dynamics, and control to produce robust and realistic robot motion.
- **Output:** Full set of executable commands — positions, velocities, accelerations, torques, and control inputs.
- **Goal:** Ensure that the robot can perform the motion safely in its actual environment, adjusting to changes and maintaining stability.

Example: Autonomous Mobile Robot in a Cluttered Environment

- The robot must navigate a warehouse with moving workers and shelves.
- It uses sensors (LiDAR, cameras) to detect obstacles in real time.
- It adjusts its trajectory dynamically based on environment updates.
- It uses feedback control to stay on course even when small disturbances occur.

Key Characteristics

- **Dynamic Feasibility:** Considers real forces, torques, and inertia.
- **Stability and Balance:** Especially important in humanoid robots or legged systems.
- **Real-Time Adaptation:** Uses sensor feedback to modify motion during execution.
- **Integration:** Combines motion planning with control and perception systems.

Common Techniques

- Model Predictive Control (MPC)
- Dynamic Window Approach (DWA)
- Feedback Linearization
- Whole-body Motion Planning for complex robots (e.g., humanoids, quadrupeds)

Scope

Motion planning is the end-to-end system that governs how the robot perceives, plans, and moves in the real world. It incorporates all stages and ensures the robot's actions are not just feasible on paper but reliable under real operational conditions.

4 Configuration Space (C-space)

What is Configuration Space?

In robotics, the term **Configuration Space** (commonly abbreviated as **C-space**) refers to the set of all possible states or configurations that a robot can assume. A configuration captures all the degrees of freedom (DoF) necessary to fully specify the pose of the robot. This includes parameters such as position, orientation, and joint angles.

Rather than planning in the physical (workspace) coordinates, motion planning is typically performed in C-space. This abstraction allows the robot's geometry and its interaction with the environment to be encoded in terms of higher-dimensional, yet more structured, variables.

Formal Definition:

Let q denote a configuration of the robot. Then, the **Configuration Space** C is defined as:

$$C = \{q \mid q \text{ is a valid configuration of the robot}\}$$

The C-space encapsulates all positions and orientations the robot can legally take, even if some are constrained or infeasible due to obstacles or kinematic restrictions.

Why Use Configuration Space?

- **Abstraction:** C-space transforms a robot with physical dimensions into a point in a higher-dimensional space. This reduces complex robot geometry and environment into manageable mathematical constructs.
- **Planning Efficiency:** Geometric planning problems like path finding become simpler when the problem is modeled in C-space.
- **Collision Checking:** Instead of checking for collision in 3D with a complex robot shape, we check whether a point in C-space lies within the *Free* or *Obstacle* subset.

A key strength of C-space is that it handles arbitrary robot geometries. For instance, whether the robot is a car-like vehicle, a multi-joint arm, or a humanoid, all these systems can be represented as points moving in a properly defined configuration space.

Examples of Configuration Spaces

- **Point Robot in 2D Plane:** Configuration: (x, y) C-space: $\mathbb{R}^2$
- **Mobile Robot with Orientation:** Configuration: (x, y, θ), where θ is heading C-space: $\mathbb{R}^2 \times S^1$
- **2-DOF Planar Manipulator:** Configuration: (θ_1, θ_2) C-space: $S^1 \times S^1$ Each joint contributes one rotational DoF.
- **6-DOF Industrial Arm:** Configuration: $(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6)$ C-space: $S^1 \times S^1 \times \cdots \times S^1$ (six times) The configuration is a point in a 6-dimensional toroidal space.
- **Humanoid Robot:** Dozens of joints with both revolute and prismatic components. C-space may exceed 30–40 dimensions depending on actuation complexity.

Dimensionality and Complexity

- As the number of DoF increases, the dimension of the C-space increases.
- Higher dimensions exponentially increase the computational complexity of planning.
- For instance, a 3-DOF robot might allow grid-based planning, but a 10-DOF robot may require sampling-based or optimization-based methods.

Case Study: 7-DOF Arm Manipulator

A 7-DOF arm (like the KUKA LBR iiwa or the Franka Emika Panda) must operate in a narrow corridor with shelves on both sides:

- Planning must occur in 7D space.
- The C-space obstacles are abstract, shaped by robot's self-collision and workspace interactions.
- Grid-based techniques are infeasible due to the curse of dimensionality.
- Sampling-based planners (like RRT) are preferred for such high-dimensional C-spaces.

Free Space and Obstacle Space

Once the configuration space C is defined, it is typically divided into two subsets:

- **Free Space (C_{free}):** The subset of C containing all configurations where the robot does not collide with any obstacles or violate constraints.

$$C_{free} = \{q \in C \mid \text{robot at } q \text{ is collision-free}\}$$

- **Obstacle Space (C_{obs}):** The subset of C where the robot is in collision with the environment, itself, or violates joint limits:

$$C_{obs} = C \setminus C_{free}$$

Why this Matters:

- Rather than modeling the robot and all obstacles in physical space, we simplify to a point navigating in a high-dimensional space.
- Path planning becomes the search for a continuous path $\gamma(t)$ such that $\gamma(t) \in C_{free}$ for all $t \in [0, 1]$.

Example: Planar Mobile Robot in an Obstacle Field

- In the 2D map, obstacles have finite size.
- In C-space, these obstacles are *grown* (via Minkowski sum) by the shape of the robot.
- The robot becomes a point; the workspace becomes C-space with inflated obstacles.
- A path is valid if the robot-point never enters C_{obs}.

Case Study: Self-Driving Car

In autonomous driving:

- The configuration includes (x, y, θ).
- C_{free} accounts for lane boundaries, other vehicles, curbs, and speed constraints.
- The car must find a trajectory that lies completely within C_{free} for safety.
- During planning, nearby cars are dynamically projected into the C-space to modify C_{obs} in real-time.

Planning in C-space

Planning in C-space allows the robot to reason abstractly about motion. While the physical workspace contains obstacles and geometry, the C-space contains constraints in terms of allowed configurations. All motion planning algorithms (A*, RRT, PRM, CHOMP, etc.) operate in C-space.

This abstraction reduces the motion planning task to a geometric search problem:

- Find a continuous curve $\gamma : [0, 1] \to C_{free}$, such that:

$$\gamma(0) = q_{start}, \quad \gamma(1) = q_{goal}$$

- The robot can then follow this path through interpolation or trajectory generation.

5 Overview of Path Planning Approaches

The Goal of Path Planning

The core objective of path planning in robotics is to compute a feasible (and often optimal) route for the robot to travel from a start configuration q_{start} to a goal configuration q_{goal} without colliding with obstacles and while satisfying any dynamic, kinematic, or environmental constraints.

Given a configuration space C, and its subsets C_{free} and C_{obs}, path planning aims to compute a continuous function $\gamma : [0,1] \rightarrow C_{\text{free}}$ such that:

$$\gamma(0) = q_{\text{start}}, \quad \gamma(1) = q_{\text{goal}}$$

Two Major Families of Approaches

Path planning methods are broadly categorized into:

1. **Search-Based Methods**
2. **Sampling-Based Methods**

Both approaches are designed to solve the same core problem, but they make different assumptions and utilize different mathematical and computational frameworks. Each has strengths suited to specific types of environments, robots, and dimensionalities.

1. Search-Based Path Planning

Concept

Search-based methods rely on discretizing the configuration space into a graph or grid structure. Each node corresponds to a discrete configuration, and edges represent allowed transitions. A classic graph search algorithm is then used to traverse this structure from the start to the goal.

Characteristics

- Deterministic: The outcome is predictable and repeatable.
- Completeness: Guarantees a solution if one exists (assuming infinite resources).
- Optimality: Many variants ensure shortest or least-cost paths.
- Precomputed structure: Requires grid or graph of the entire environment.

Common Algorithms

- **Breadth-First Search (BFS):** Explores nodes level by level. Guarantees shortest path (in terms of steps) but inefficient in large spaces.
- **Dijkstra's Algorithm:** Computes shortest path based on actual travel cost. Explores all possible paths until goal is reached. Guarantees optimality but can be slow in high dimensions.
- **A* (A-Star) Search:** An extension of Dijkstra's algorithm using a heuristic to guide the search toward the goal more efficiently. Common in mobile robot navigation.
- **D* and D* Lite:** Designed for dynamic environments. Efficiently updates the shortest path as the robot discovers new obstacles or map changes.

Example: Grid-Based Navigation

- Workspace is divided into a 2D grid (like pixels in an image).
- Robot moves from cell to cell (e.g., 4-connected or 8-connected).
- Obstacles are marked as impassable cells.
- A* is used to compute shortest path from start to goal.
- Heuristic: Euclidean or Manhattan distance.

Case Study: Indoor Delivery Robot

A wheeled robot navigating an office floor with known map:

- Preloaded occupancy grid.
- A* used to find paths to rooms.
- Low-level controllers handle local movement and obstacle avoidance.
- Grid updates dynamically with temporary obstacles (e.g., humans).

Limitations of Search-Based Methods:

- Curse of dimensionality: Grids become exponentially large as DoF increases.
- Requires explicit discretization, which may miss narrow passages.
- Not suitable for real-time planning in continuous or high-dimensional spaces.

2. Sampling-Based Path Planning

Concept

Sampling-based algorithms bypass the need for discretization. Instead, they randomly sample configurations from the C-space and attempt to connect them to construct a graph (roadmap) or tree representing feasible paths. These methods do not explicitly construct the obstacle space; instead, they check collisions only when needed.

Characteristics

- Probabilistic Completeness: If a path exists, probability of finding it approaches 1 as samples increase.
- Suited for high-dimensional spaces.
- No need to construct full configuration space or workspace.
- Trade-off: May not guarantee optimality unless modified (e.g., RRT*).

Common Algorithms

- **Probabilistic Roadmap (PRM):**
  - Phase 1: Learning — Sample many configurations and connect neighbors to form a roadmap.
  - Phase 2: Query — Connect start and goal to the roadmap and find shortest path.
  - Best for static, multi-query environments (e.g., warehouse robots).
- **Rapidly-Exploring Random Tree (RRT):**
  - Builds a tree incrementally from the start configuration toward the goal.
  - Bi-directional RRT builds two trees simultaneously.
  - Good for single-query, high-dimensional spaces.
- **RRT*:** A variant of RRT that guarantees asymptotic optimality by rewiring the tree to minimize path cost over time.

Example: 7-DOF Manipulator Planning

- Workspace has tight spaces between obstacles (e.g., shelves).
- C-space has 7 dimensions; grid methods are computationally infeasible.
- RRT samples valid joint configurations and builds tree from initial arm pose.
- Local planner ensures that motion between nodes is collision-free.

Case Study: Mars Rover Manipulator Arm

Planning for sample collection with multiple joints and constraints:

- PRM is precomputed on Earth based on terrain map.
- Queries onboard select path to desired sample sites.
- Reduces real-time computation and handles high DoF with collision constraints.

Limitations of Sampling-Based Methods:

- No completeness or optimality guarantee unless special variants are used.
- Narrow passages are difficult to discover via random sampling.
- Path smoothness may be poor unless post-processing is applied.

Choosing the Right Method

- **Known Static Environment + Low DoF:** Use A*, Dijkstra.
- **Dynamic Environment:** Use D*, D* Lite.
- **High-Dimensional Space:** Use RRT, PRM.
- **Need Optimality:** Use A* (search-based) or RRT* (sampling-based).
- **Multi-query setting:** Prefer PRM.

6 Search-Based Algorithms for Path Planning

Search-based algorithms are a foundational class of motion planning techniques that treat the environment as a graph or grid and search for a path from a start node to a goal node. These methods operate by systematically exploring the space of possible configurations using well-defined rules.

They are characterized by:

- Discretization of the configuration or workspace into a structured graph.
- Use of search heuristics or cost functions to guide exploration.
- Guarantees of completeness and/or optimality depending on the algorithm.

In this section, we study four widely used search-based methods:

- Breadth-First Search (BFS)
- Dijkstra's Algorithm
- Greedy Best-First Search
- A* Search

Breadth-First Search (BFS)

Core Idea: BFS explores all nodes at the current depth before moving to the next depth level. It treats the environment as an unweighted graph where each step has equal cost.

- The frontier is a First-In-First-Out (FIFO) queue.
- At each step, the node at the front of the queue is expanded.
- All unvisited neighbors are added to the back of the queue.

Properties:

- **Completeness:** Yes — BFS will always find a path if one exists.
- **Optimality:** Yes, but only in unweighted graphs where all edge costs are equal.
- **Time Complexity:** $\mathcal{O}(b^d)$ where b is branching factor, d is the depth of the shallowest solution.
- **Space Complexity:** $\mathcal{O}(b^d)$ — stores all nodes in the queue.

Example:

- Grid world of 10×10.
- Robot starts at (0,0), goal is at (9,9).
- Obstacles are static and sparse.
- BFS finds the shortest number of steps, but explores large parts of the grid.

Use Case:

- Educational environments.
- Simple mobile robot navigation with uniform terrain.

Dijkstra's Algorithm

Core Idea: Extends BFS to handle weighted graphs. Instead of layers, it expands the node with the lowest cumulative cost-from-start $g(n)$.

- Maintains a priority queue sorted by $g(n)$.
- At each step, the node with the smallest $g(n)$ is expanded.
- Neighbors are added/updated in the queue with accumulated cost.

Properties:

- **Completeness:** Yes — will always find a path if one exists.
- **Optimality:** Yes — always returns the least-cost path.
- **Heuristic:** None — does not use goal-based information.
- **Time Complexity:** $\mathcal{O}(|E| + |V| \log |V|)$ using min-heap.

Example:

- Robot navigating a terrain with different cost values (e.g., sand = 5, grass = 1).
- Dijkstra accounts for these costs and avoids high-cost paths.

Case Study:

- Delivery drone flying in city — each air corridor has energy cost.
- Dijkstra finds energy-efficient route, not necessarily fastest.

Greedy Best-First Search

Core Idea: Uses a heuristic function $h(n)$ to expand the node that *appears* closest to the goal. Ignores the cost-so-far.

- Maintains a priority queue sorted by $h(n)$.
- At each step, expands the node with smallest estimated cost to goal.

Properties:

- **Completeness:** Not guaranteed — can get stuck in local minima.
- **Optimality:** No — does not consider actual path cost.
- **Heuristic:** Required.
- **Time Complexity:** Depends on heuristic quality.

Example:

- Robot moves toward goal based on Euclidean distance.
- May end up in dead-end if heuristic does not account for obstacles.

Use Case:

- Time-critical applications where path quality is less important than speed.
- Game AI movement (e.g., enemies chasing player).

A* Search

Core Idea: Combines Dijkstra and Greedy BFS by balancing actual cost-so-far $g(n)$ and heuristic estimate $h(n)$:

$$f(n) = g(n) + h(n)$$

- Maintains a priority queue sorted by $f(n)$.
- Explores nodes that minimize the combined estimated cost.

Heuristic Requirements:

- **Admissible:** Never overestimates actual cost to goal.
- **Consistent (Monotonic):** $h(n) \leq c(n, n') + h(n')$ for all edges.

Properties:

- **Completeness:** Yes, if heuristic is admissible and space is finite.
- **Optimality:** Yes, with admissible heuristic.
- **Time Complexity:** Depends on heuristic — $\mathcal{O}(b^d)$ in worst case.
- **Space Complexity:** Can be high — stores all explored nodes.

Example:

- Self-driving car planning in a city.
- $g(n)$ is time or energy used so far.
- $h(n)$ is estimated remaining time using road speed limits.
- A* finds quickest valid path.

Case Study: Mobile Robot in Hospital

- Robot must deliver medicines from storage to ICU.
- Uses A* with $g(n)$ = distance travelled and $h(n)$ = Euclidean distance to ICU.
- Ensures safety and efficiency through optimal route.
- Real-time re-planning possible if corridors become blocked.

7 A* Algorithm: Detailed Exploration in Motion Planning

The A* (pronounced "A-star") search algorithm is one of the most influential and widely used search techniques in the fields of robotics, computer science, and artificial intelligence. It elegantly merges two competing objectives: cost minimization (as in Dijkstra's algorithm) and intelligent directionality (as in heuristic-based search like Greedy BFS). Due to this balance, A* has become the gold standard for optimal path planning in environments with obstacles, especially where computational efficiency matters.

Why A* is Widely Adopted

A* finds widespread use in a range of robotic and non-robotic systems:

- **Robotics Path Planning:** Mobile robots, UAVs, warehouse robots, and planetary rovers employ A* for safe and efficient navigation.
- **Video Game AI:** A* powers enemy pursuit, terrain traversal, and strategy execution in real-time environments.
- **GPS and Navigation Systems:** A* helps suggest optimal travel routes considering real-world constraints like traffic or roadblocks.
- **Indoor Navigation:** Wheelchair robots, hospital trolleys, and domestic bots rely on A* to navigate dynamic layouts.

The key reason for A*'s popularity is that it is both **complete** (it finds a solution if one exists) and **optimally efficient** (no other optimal algorithm is guaranteed to expand fewer nodes).

The Core Principle of A*

A* operates on the evaluation function:

$$f(n) = g(n) + h(n)$$

Where:

- $g(n)$: The cumulative cost to reach node n from the start node. This represents **what has already been spent**.
- $h(n)$: The heuristic estimate of the cost from node n to the goal. This captures **what still needs to be spent**.

At each iteration, the algorithm selects the node n from the open list that minimizes $f(n)$. This balances:

- **Exploration:** By considering $g(n)$, A* avoids paths that are unnecessarily long.
- **Exploitation:** By considering $h(n)$, A* prioritizes directions that look promising toward the goal.

Detailed Step-by-Step Execution of A*

Let us now walk through the typical procedure A* follows in a discrete environment such as a grid or graph:

1. **Initialize:**
   - Place the start node in the open list (a priority queue based on $f(n)$).
   - The closed list (set of visited nodes) is initially empty.
2. **Main Loop:**
   - While the open list is not empty:
     (a) Extract the node q with the lowest $f(q)$ from the open list.
     (b) If q is the goal node, terminate. Reconstruct path from parent links.

(c) Otherwise, expand q:
   - Generate all valid neighbors of q.
   - For each neighbor n:
      * Compute $g(n)$, $h(n)$, and $f(n)$.
      * If n is not in open or closed list, add it.
      * If n is in open list but this path has a lower $g(n)$, update it.
      * If n is in closed list and new path is better, move it back to open.

(d) Move q to the closed list.

Implementation Details:

- Each node stores parent pointers for path reconstruction.
- The open list is typically implemented using a min-heap.
- Hash tables are used for fast access to the closed list.

Heuristics Used in A*

The efficiency and optimality of A* heavily rely on the heuristic function $h(n)$. It should be:

- **Admissible:** $h(n) \leq h^*(n)$ for all n, where h^* is the true minimal cost-to-goal.
- **Consistent (Monotonic):** $h(n) \leq c(n, n') + h(n')$, ensuring A* never revisits nodes unnecessarily.

Common Heuristics:

1. **Manhattan Distance:**
$$h(n) = |x_n - x_g| + |y_n - y_g|$$
Appropriate for grid-based motion with 4-directional moves (no diagonals).
2. **Diagonal Distance:**
$$h(n) = \max(|x_n - x_g|, |y_n - y_g|)$$
Used when diagonal movements are allowed (e.g., 8-connected grid).
3. **Euclidean Distance:**
$$h(n) = \sqrt{(x_n - x_g)^2 + (y_n - y_g)^2}$$
Applicable when motion is unrestricted in continuous space.

Examples and Case Studies

Example 1: Maze Navigation A 2D grid robot needs to reach the bottom-right corner from the top-left corner. Obstacles are randomly placed. The Manhattan heuristic is used.

- A* explores the grid in a wavefront toward the goal.
- Avoids all obstacles while taking the shortest possible route.

Example 2: Urban Autonomous Car An autonomous car in a smart city needs to navigate from Sector 21 to Sector 44.

- $g(n)$ = time taken on road segments (varies with traffic).
- $h(n)$ = straight-line (Euclidean) distance to goal.
- Roads with heavy traffic have high $g(n)$.
- A* finds the path balancing travel distance and congestion.

Example 3: Warehouse Robot A warehouse robot must fetch a package from a distant shelf avoiding other moving robots and static shelves.

- Environment is a grid with dynamic updates.
- $h(n)$ is calculated using precomputed distance maps.
- $g(n)$ includes energy cost due to acceleration/deceleration.
- A* recalculates path periodically to adjust for moving robots.

Relation to Other Algorithms

- **Dijkstra's Algorithm:** Equivalent to A* with $h(n) = 0$ for all nodes. Explores extensively.
- **Greedy Best-First Search:** Equivalent to A* with $g(n) = 0$. Focuses only on heuristic; may miss optimal paths.
- **A* as a Generalization:** Balances both cost-so-far and goal-direction, adapting behavior through $h(n)$.

8 Sampling-Based Algorithms: Rapidly-exploring Random Trees (RRT)

Traditional search-based methods such as A* or Dijkstra's algorithm tend to perform poorly in high-dimensional spaces. As the number of dimensions increases—whether due to joint angles in a robotic manipulator, or 6-DOF motion of aerial drones—the computational cost of discretizing the configuration space grows exponentially, a phenomenon known as the **curse of dimensionality**. In such scenarios, **sampling-based planning methods** offer a scalable and practical solution.

Why Sampling-Based Planning?

Sampling-based planners overcome the limitations of discrete graph-based methods by avoiding complete enumeration of the configuration space. Instead of exploring every node on a pre-defined grid, they:

- Sample points from the configuration space directly and randomly.
- Build data structures (e.g., trees or graphs) incrementally.
- Bypass the need for explicit representation of the entire space.

Benefits:

- Suitable for **high-dimensional systems** (e.g., 7-DOF arms, legged robots).
- Efficient in **complex, cluttered environments** with narrow passages.
- Naturally accommodate **continuous, dynamic** spaces.
- Highly adaptable to different robot types and motion constraints.

Types of Sampling-Based Planners

Sampling-based algorithms generally fall into two categories:

1. **Roadmap-Based Methods (Multi-query planners):**
    - Build a graph of feasible configurations (nodes) and collision-free paths (edges).
    - Used when the environment is static and multiple planning queries are expected.
    - **Example:** Probabilistic Roadmaps (PRM).
2. **Tree-Based Methods (Single-query planners):**
    - Start from the initial configuration and grow a tree toward the goal.
    - Suitable for dynamic queries or changing goals.
    - **Example:** Rapidly-exploring Random Trees (RRT).

Motivation Behind RRT

The RRT algorithm was developed with the goal of efficiently exploring large, high-dimensional configuration spaces. Its core strength lies in the ability to rapidly and systematically explore regions that are far from existing samples, which allows it to "spread out" quickly across the C-space.

Ideal Use Cases:

- **Articulated robot arms:** with many degrees of freedom and complex constraints.
- **Non-holonomic systems:** like cars, wheeled mobile robots, or drones with motion constraints.
- **Dynamic and partially known environments:** where real-time replanning is necessary.
- **Kinodynamic Planning:** where both kinematics and dynamics are enforced.

RRT: Core Intuition

The central idea in RRT is to bias exploration toward **unexplored areas** of the configuration space. This happens naturally because random samples are more likely to fall in unexplored regions. The result is a tree structure that grows outward from the initial configuration, reaching out like tendrils through the space.

RRT Algorithm: Step-by-Step Procedure

Given an initial configuration x_{init}, the RRT builds a tree rooted at x_{init} and expands it toward randomly sampled configurations. The main steps are:

1. **Sample:** Generate a random configuration x_{rand} from the configuration space.
2. **Nearest Neighbor:** Find the nearest node x_{near} in the tree to x_{rand}, typically using Euclidean or weighted distance metrics.
3. **Steer:** Move from x_{near} toward x_{rand} by a small step size λ to create a new configuration x_{new}.
4. **Collision Check:** If the motion from x_{near} to x_{new} is collision-free (i.e., it lies within C_{free}), then:
    - Add x_{new} to the tree as a new node.
    - Add an edge between x_{near} and x_{new}.
5. **Check Goal:** If x_{new} is sufficiently close to the goal configuration, stop the algorithm and return the path from root to goal.
6. **Repeat:** Otherwise, continue the expansion for a fixed number of iterations or until a timeout.

Example: 6-DOF Robotic Arm

Imagine a 6-joint industrial robot arm that must navigate through a cluttered workspace to weld a joint.

- The configuration space is 6-dimensional: $(\theta_1, \theta_2, ..., \theta_6)$.
- Traditional grid-based planning becomes computationally infeasible.
- RRT samples joint configurations randomly and steers from current tree toward them.
- The arm avoids self-collisions and environmental obstacles.
- A path is generated without full discretization of each joint space.

Algorithmic Pseudocode

```
Function RRT(x_init, x_goal):
    T.init(x_init)
    for i = 1 to N:
        x_rand ← Sample()
        x_near ← Nearest(T, x_rand)
        x_new ← Steer(x_near, x_rand, )
        if CollisionFree(x_near, x_new):
            T.add_node(x_new)
            T.add_edge(x_near, x_new)
            if Distance(x_new, x_goal) < :
                return Path from x_init to x_goal
    return Failure
```

Limitations of RRT

While RRT is highly efficient and practical, it has several limitations:

- **Suboptimal Paths:** The tree does not guarantee shortest or smoothest paths.
- **Lack of Completeness:** RRT is probabilistically complete—not guaranteed to find a path, but probability increases with more samples.
- **Resolution Sensitivity:** The step size λ must be tuned carefully.
- **Bias in Sampling:** Random sampling may miss narrow passages unless strategies are used.

Extensions and Variants

Numerous enhancements exist to address RRT's shortcomings:

- **RRT*:** Asymptotically optimal version that rewires the tree to minimize path cost.
- **Bidirectional RRT:** Grows two trees (from start and goal) and connects them.
- **Kinodynamic RRT:** Accounts for robot dynamics and velocity constraints.
- **RRT-Connect:** Rapidly grows trees by extending aggressively toward sampled points.

9 RRT Algorithm: Worked Example

To understand the working of the RRT algorithm in practice, let us walk through two iterations in a simple 2D configuration space.

Configuration Space Setup

- **Configuration Space (C-space):** $[0, 10] \times [0, 10]$ — a continuous 2D plane.
- **Start Configuration:** $(1, 1)$
- **Goal Configuration:** $(9, 9)$
- **Step Size:** $\lambda = 1$

Iteration 1

- **Random Sample:** $x_{\text{rand}} = (3, 8)$
- **Tree Nodes (initially):** $\{(1, 1)\}$
- **Find Nearest Node:**

$$x_{\text{near}} = (1, 1)$$

- **Direction Vector:**
$$\Delta x = (3-1, 8-1) = (2, 7)$$
- **Distance:**
$$\|\Delta x\| = \sqrt{2^2 + 7^2} = \sqrt{4+49} = \sqrt{53} \approx 7.28$$
- **Unit Direction Vector:**
$$\hat{u} = \left(\frac{2}{7.28}, \frac{7}{7.28}\right) \approx (0.274, 0.961)$$
- **New Node (1-step in direction):**
$$x_{\text{new}} = (1,1) + \lambda \cdot \hat{u} = (1+0.274, 1+0.961) = (1.274, 1.961)$$
- **Collision Check:** The path from $(1,1)$ to $(1.274, 1.961)$ lies in C_{free}.
- **Update Tree:**
$$\text{Tree Nodes} \leftarrow \{(1,1), (1.274, 1.961)\}$$
$$\text{Tree Edges} \leftarrow \{((1,1) \to (1.274, 1.961))\}$$

Iteration 2

- **Random Sample:** $x_{\text{rand}} = (5, 5)$
- **Tree Nodes:** $\{(1,1), (1.274, 1.961)\}$
- **Find Nearest Node:**
  - Distance to $(1,1)$:
  $$\sqrt{(5-1)^2 + (5-1)^2} = \sqrt{16+16} = \sqrt{32} \approx 5.656$$
  - Distance to $(1.274, 1.961)$:
  $$\Delta x = (5 - 1.274, 5 - 1.961) = (3.726, 3.039)$$
  $$\|\Delta x\| = \sqrt{3.726^2 + 3.039^2} \approx \sqrt{13.88 + 9.24} = \sqrt{23.12} \approx 4.808$$
  - **Nearest Node:** $(1.274, 1.961)$
- **Direction Vector:**
$$\Delta x = (3.726, 3.039)$$
- **Unit Direction Vector:**
$$\hat{u} = \left(\frac{3.726}{4.808}, \frac{3.039}{4.808}\right) \approx (0.774, 0.632)$$
- **New Node:**
$$x_{\text{new}} = (1.274 + 0.774, 1.961 + 0.632) = (2.048, 2.593)$$
- **Collision Check:** The segment from $(1.274, 1.961)$ to $(2.048, 2.593)$ is in C_{free}.
- **Update Tree:**
$$\text{Tree Nodes} \leftarrow \{(1,1), (1.274, 1.961), (2.048, 2.593)\}$$
$$\text{Tree Edges} \leftarrow \{((1,1) \to (1.274, 1.961)), ((1.274, 1.961) \to (2.048, 2.593))\}$$

Observation

- The RRT incrementally builds a tree, exploring outward from the start configuration.
- Each iteration adds a new node approximately $\lambda = 1$ unit away from the nearest existing node in the direction of a randomly sampled point.
- With enough iterations, the tree will eventually reach near the goal region, even in cluttered or high-dimensional environments.

10 RRT Algorithm: Pseudocode and Properties

Pseudocode: Rapidly-Exploring Random Tree (RRT)

- **Input:**
  - x_{init}: Start configuration
  - x_{goal}: Goal configuration
  - λ: Step size
  - $N_{\max}$: Maximum number of iterations
- **Initialize:**
  - Tree $T \leftarrow \{x_{\text{init}}\}$ with root at start node
- **Loop:** For $i = 1$ to $N_{\max}$:
  1. Sample a random configuration x_{rand} from the configuration space.
  2. Find the nearest node in the tree T, denoted x_{near}.
  3. Move from x_{near} toward x_{rand} by a step size λ to compute x_{new}.
  4. **Collision Check:** If the path from x_{near} to x_{new} lies in C_{free}:
     - Add x_{new} to the set of nodes in T.
     - Add edge $(x_{\text{near}} \to x_{\text{new}})$ to T.
  5. **Goal Check:** If x_{new} is within a small threshold distance from x_{goal}:
     - **Terminate:** A path to goal has been found.
- **Failure:** If goal is not reached after $N_{\max}$ iterations:
  - Return failure.

Properties of RRT

Strengths

- Well-suited for high-dimensional configuration spaces.
- Efficiently covers large unexplored areas due to inherent sampling bias.
- Handles complex environments and continuous motion.
- Simple and computationally fast to implement.

Limitations

- Does **not guarantee optimal paths**; resulting trajectories may be long or jagged.
- Often requires post-processing (e.g., path smoothing).
- **Randomized nature** means that different executions may produce different results.
- **Probabilistic completeness:** A solution is found only with high probability given infinite time; no guarantee in finite time.

Extensions

- **RRT*:** An asymptotically optimal version of RRT that rewires the tree for better path quality over time.
- **Kinodynamic RRT:** Considers robot dynamics and non-holonomic constraints (e.g., cars, drones).