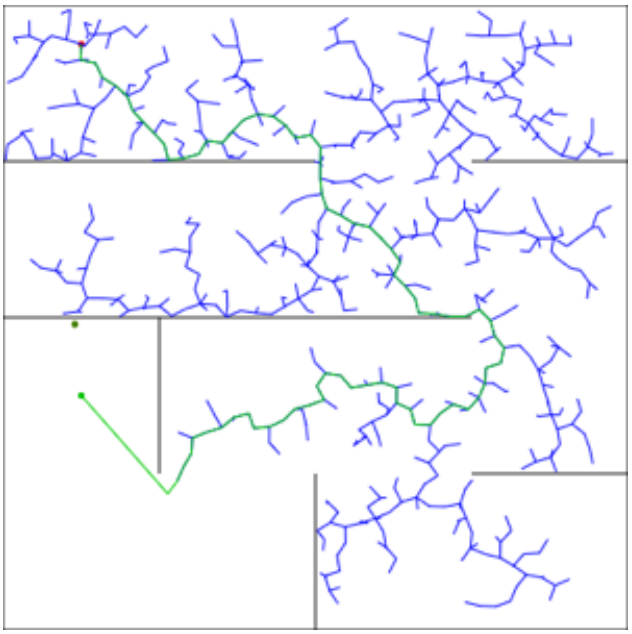
Project: Rapidly-Exploring Random Trees

1 Background

The *Path Planning* problem involves finding a path from a start to a goal point that do not collide with obstacles. This is a fundamental problem that every mobile robot must be able to solve. In this project, you will implement an algorithm called *Rapidly-Exploring Random Tree (RRT)* to plan for collision-free paths in 2-dimensional space. The figure below shows an example result: the red dot indicates the start location, the green dot the goal location, the blue lines the built tree, and the green lines the obstacle-free path found.



The first line of your program must be the following:

```
let r = require('rrt');
```

This line loads a library of geometry functions, visualization tools, and example maps.

2 Provided Geometry Classes and Functions

We have provided several geometry helper functions and classes that you will need for this project. The `Point` class represents a two-dimensional point:

```
let p = new r.Point(12, 34); // produces { x: 12, y: 34 }
```

The `Line` class represent a line segment:

```
let l = new r.Line(new r.Point(1, 2), new r.Point(3, 4));
// produces { p1: { x: 1, y: 2 }, p2: { x: 3, y: 4 } }
```

The `intersects` function determines whether two line segments intersect:

```
r.intersects(line1: r.Line, line2: r.Line): boolean
```

3 Maps

The map in which the RRT planner will work on is defined by two things: its size and the obstacles inside of it. All the maps in this project will have a square shape: their height and width will be equal. Thus, the size of a map can be represented by a single numerical variable, named mapSize which will reside in the options object (see the following section).

The walls, or obstacles, of the map are represented by a single array containing several Line objects. Each of this objects corresponds to exactly one wall inside the map. The `rrt` library has two maps that you can use for testing, and a function that generates random hard maps:

```
r.getDemoMap() : { size: number, lines: Line[] }
r.getIrregularObstacleMap(): { size: number, lines: Line[] }
r.generateHardMap(size: number, spacing: number,
                  opening: number): { size: number, lines: Line[] }
```

You can visualize these maps using a drawing canvas:

```
let canvas = lib220.newCanvas(400,400)
r.drawMap(canvas, r.getDemoMap());
```

4 Programming Task

Required Helper Functions You will need to implement the following helper functions:

- `distance(p1: Point, p2: Point): number`

  `distance` calculates the distance between two points.

- `samplePoint(mapSize: number, goal: Point, goalBias: number): Point`

  `samplePoint` returns either (1) a randomly sampled 2D point within the map or (2) the `goal`. The function randomly chooses between the two outcomes, based on the value of `goalBias`. When `goalBias === 1`, it always produces `goal` and when `goalBias === 0` it always returns a randomly sampled point.

- `collides(map: Line[], p1: Point, p2: Point): boolean`

  `collides` returns true if the line from `p1` to `p2` intersects of the lines in `map`.

- `getPath(goal: Tree): Point[]`

  `getPath` takes a subtree with the goal as the node and returns the path to the starting point. The first point in the array must be the goal, and the last point the start point.

Representing the Random Tree Implement the `Tree` class (sketched in fig. 1), which will be used to implement the RRT algorithm. The class has the following methods:

- `tree.nearest(p)` returns the root of a subtree that is closest to `p`.
- `tree.extend(p, maxExtension)` produces a point `r`. If the distance between `p` and `tree.node` is less than `maxExtension` then `r` should be `p`. Otherwise, `r` should be the point on the line between `tree.node` and `p`, such that `distance(tree.node, r) === maxExtension`.
- `tree.add(p)` adds a new immediate subtree to `tree` with `p` as the root node and no sub-children.

```
class Tree {
  // constructor(node: Point, children: Tree[], parent: Tree | null)
  constructor(node, children, parent) {
    this.node = node;
    this.children = children;
    this.parent = parent;
  }

  // nearest(p : Point): Tree
  nearest(p) {
    ...
  }

  // extend(p: Point, maxExtension: number): Point
  extend(p, maxExtension) {
    ...
  }

  // add(p: Point): Tree
  add(p) {
    ...
  }
}
```

Figure 1: Outline of the `Tree` class.

Putting Everything Together: The RRT Planner Define the following function to implement the RRT algorithm:

```
type Options = {
    mapSize: number,
    maxExtension: number,
    goalBias: number,
    maxSamples: number,
    callback: (p: Point, q: Tree, r: Point, t: Tree) => void
};

plan(start: Point, goal: Point, map: Line[], options: Options): Point[] | undefined,
```

The `options` object has several parameters that affect the algorithm, most of which have been explained above. The two new parameters are:

- `maxSamples`: The maximum number of samples that the RRT planner should generate. If the planner does not find the goal after `maxSamples` steps, it should produce `undefined` instead of a path from the goal to the start point.
- `callback`: A callback function that must be called at every iteration of the RRT algorithm, with the following parameters
  - p, the current sampled point
  - q, the closest leaf in the tree to p
  - r, the extension point from q to r
  - t, the current tree grown so far.

5 Design decisions, debugging, and visualizations

You will have to make several design decisions to successfully implement the RRT algorithm. Here are some points you will have to think about:

- The callback function in the options object can be used to visualize the progress of the RRT algorithm, and to visually inspect for correctness. How can you implement this function? Note that the lines in a map do not have units. For visualization purposes, you can assume the point positions use the same units as the pixels in a canvas.
- The goalBias and maxExtension values vary how the RRT algorithm performs on different maps. While we will not be evaluating your implementation for efficiency, you can play around with their values to see how it affects the resulting tree generated, and the final path found.

6 Visualization

To aid in visualization, there are a number of drawing functions that are part of the `rrt` library. To visualize the tree, you should make use of the callback function located in the options object. If you want to visualize the tree as it is created step by step, you will want to use the `lib220.sleep(n)` function which will pause your program for n milliseconds.

```
type Color = [ number, number, number ];
```

A color is represented as an array of three numbers, where each number is in the range 0 to 1. The numbers indicate the intensity of red, green, and blue respectively.

```
r.newCanvas(width: number, height: number): Canvas
```

Creates a blank white canvas and displays it in Ocelot.

```
Canvas.drawLine(x1: number, y1: number, x2: number, y2: number, color: Color): void
```

Draws a line on the canvas from (x_1, y_1) to (x_2, y_2).

```
Canvas.drawArc(x: number, y: number, radius: number, startRadians: number,
               endRadians: number, color: Color): void
```

Draws an arc centered at (x, y), where the curve starts at `startRadians` and ends at `endRadians`.

```
Canvas.drawCircle(x: number, y: number, radius: number, color: Color): void
```

Draws a circle.

```
Canvas.drawFilledCircle(x: number, y: number, radius: number, color: Color): void
```

Draws a filled circle.

```
Canvas.clear();
```

Clears the canvas.