Algorithmic Robotics
COMP/ELEC/MECH 450/550
Project 1: Getting Familiar with OMPL - C++ Version

This project must be completed individually. Submissions are due September 18 at 1:00pm via Canvas.

The documentation for OMPL can be found at http://ompl.kavrakilab.org.

In this project, you will familiarize yourself with the basics of compiling and running programs that use the Open Motion Planning Library (OMPL) C++ library. You will run programs that execute basic motion planning queries for a variety of systems, run motion planning queries for a robotic arm in PyBullet, and do basic evaluation of sampling-based planners through OMPL's benchmarking interface. Although there is minimal code to write, it is expected that you compile and run OMPL C++ programs, inspect and execute demo code, and evaluate interesting planning scenarios using the provided programs. **Note: the second part of this project may take a while to run. Please start early and plan accordingly.**

We will assume in the project documentation that you are using the provided Docker environment or have installed OMPL C++ libraries on your system. Adapt any statements as necessary to fit your installation.

Project Exercises

Getting Started

Step 1: Copy Project into Docker Container

Copy your project files into the Docker container:

```
docker cp ./project1 comp450-workspace:/workspace/
```

Step 2: Start Container and Build

Start your container and enter it:

```
docker start comp450-workspace
docker exec -it comp450-workspace bash
```

Navigate to your project and build:

```
cd /workspace/project1
make install-deps-docker
make all
```

Or build individual components:

```
make rigid_body_planning
make project1
make benchmarking
```

Step 3: Copy Results Out of Container

After running experiments, copy results back to your local machine:

```
docker cp comp450-workspace:/workspace/project1/trajectory_video.mp4 ./
docker cp comp450-workspace:/workspace/project1/*.log ./
docker cp comp450-workspace:/workspace/project1/benchmark.db ./
```

1. Running Example Programs

The goal of this part is to understand how to run different planners in OMPL using C++. In this exercise, we want you to get a feel for how problems are constructed using OMPL C++ as well as how difficult certain problems are.

1.1. First, you should run the following programs for rigid body planning. There is no visualization provided for these two programs. The goal is to make sure that you understand how to call various planners in C++. It will be helpful for your assignments later.

1. `rigid_body_planning`: Planning for a free-flying rigid body in SE(2) with obstacles.
2. `rigid_body_planning_controls`: Planning for a car-like system using first-order controls (steering velocity and forward velocity) with no obstacles.

Run these programs:

```
./rigid_body_planning
./rigid_body_planning_controls
```

Inspect the source code `src/RigidBodyPlanning.cpp` and `src/RigidBodyPlanningWithControls.cpp` to see how to setup and call a planner to solve motion planning problems in C++.

1.2. The program `project1` runs several planners in OMPL on a UR-5 robot in the PyBullet simulator. The C++ version uses a simplified problem setup with default configurations for demonstration purposes.

First, inspect the source code `src/project1.cpp`, `src/ManipulatorPlanner.cpp`, and the header files to see how motion planning problems for a robotic manipulator are constructed using OMPL C++.

Next, you should run the `project1` executable with different options. You need to run it on at least one instance of each of the **seven** different scenes (see `--problem` below). Inspect the terminal outputs and the generated solution paths.

After running the planner, you can generate MP4 videos of the robot executing the solution using the provided visualization script:

```
# First run the planner
./project1 --robot=ur5 --planner=RRTConnect --problem=bookshelf_small --index=1

# Then generate a video of the solution
python3 visualize_solution.py --robot=ur5 --problem=bookshelf_small --index=1
```

The video generator creates MP4 files showing the robot moving through the environment following the planned path. This is useful for understanding the robot's motion and verifying that solutions are correct.

The available options are:

- `--robot`: The available options are `ur5` and `fetch`.
- `--problem`: This option lets you plan on different scenes. The available options are `bookshelf_tall`, `bookshelf_thin`, `table_under_pick`, `bookshelf_small`, `box`, `table_pick`, and `cage`.
- `--index`: Each scene has different randomly-generated problem instances. You can specify the problem number with this option (the C++ version uses simplified default configurations).
- `--planner`: This specifies the motion planner you want to plan with. The available options are `PRM`, `RRT`, `RRTConnect`, and `KPIECE`. These are all very popular planners that we will discuss later in class in detail. For this assignment, you do not need to know the details of the planners.
- `--planner_range`: This is a planner parameter you can set for planners like `RRT`, `RRTConnect`, and `KPIECE`. Attempting to set this for `PRM` will have no effect. Default value is used if you do not specify this.
- `--goal_bias`: This is a planner parameter you can set for planners like `RRT`, and `KPIECE`. Default value is used if you do not specify this.
- `--max_nn`: This is a planner parameter you can set for planners like `PRM`. Default value is used if you do not specify this.
- `--planning_time`: Maximum planning time in seconds.

Video Generation Options:

The `visualize_solution.py` script provides several options for generating videos:

- `--output`: Output video filename (default: `trajectory_video.mp4`)
- `--fps`: Video frame rate (default: 10 FPS)
- `--width` and `--height`: Video resolution (default: 1280x720)
- `--start-goal-only`: Generate video showing only start and goal configurations

Example video generation commands:

```
# Basic video generation
python3 visualize_solution.py --robot=ur5 --problem=table_pick --index=1

# High-resolution video with custom filename
python3 visualize_solution.py --robot=ur5 --problem=cage --index=2 \
  --output=cage_solution.mp4 --width=1920 --height=1080 --fps=30

# Show only start and goal positions
python3 visualize_solution.py --robot=ur5 --problem=box --index=1 --start-goal-
    only
```

Below is an example if you want to run `RRTConnect` on scene `table_pick` with range set to be 0.5 and goal bias 0.01.

```
./project1 --problem=table_pick --planner=RRTConnect --planner_range=0.5 --
    goal_bias=0.01
```

You will find that each time you call the same planner on the same problem, the output path, the computation time, and the number of created states etc., vary. This is because these planners are by nature random (you will learn later in the class). Please call the above programs multiple times to observe this. You will learn how to evaluate and compare the performance of different planners in the next section.

2. Benchmarking Motion Planners

Benchmarking sampling-based planners is critical to understanding how they perform against any metric. Sampling-based planners are by nature random, and thus can have a wide range of performance purely based on luck of the draw. In this exercise, you will understand how to benchmark OMPL planners and understand their output. You will benchmark a set of planners on the following two different problems.

1. Robot `ur5`, scene `bookshelf_small`, problem index 1.
2. Robot `fetch`, scene `table_pick`, problem index 1.

We provide a benchmarking program `benchmarking` which is based on the benchmarking tool developed in OMPL. This program specifies a set of motion planners with different parameters, and runs them on the same problem instance 50 times with certain time and memory limits.

2.1. Inspect `src/benchmarking.cpp` to get a feel of how benchmarking is done with sampling-based motion planners in C++.

2.2. The standard benchmarking process with OMPL is: 1) correctly set up the benchmarking program. 2) run the benchmarking program to get data in log files. 3) run a post-processing script to convert a log file to a database file. 4) upload the database file to Planner Arena (http://plannerarena.org/) which helps you to visualize the data. You will be asked to benchmark on two problems. Detailed instructions are provided in sections **2.3** and **2.4**.

2.3. Run the provided benchmarking program on the scene `bookshelf_small` with robot `ur5`. It will generate several files, including a log file that contains your data.

```
./benchmarking --problem=bookshelf_small --robot=ur5
```

These may take a while to run (benchmarking a single problem may take approximately an hour, with maximum 3.5 hours to finish). If you find you have to increase the planning timeout to get meaningful results, this will take even longer. Please plan accordingly, in case you need to re-run any problem. After completing the benchmark, a set of `*.log` files will be created in the directory. These contain the output of the benchmarking runs.

2.4. Visualize your benchmarking data. We will be using Planner Arena (http://plannerarena.org/), a website for interactive visualization of benchmarking data gathered from OMPL benchmarking. We provide a script that processes the log file into a database file. Run the following script in order to generate a database of planning results, `benchmark.db`. Replace `*.log` with your generated log file's name.

```
python3 ompl_benchmark_statistics.py *.log
```

Note: The script `ompl_benchmark_statistics.py` will produce box plots for continuously-valued performance metrics. If you are unfamiliar with these plots, Wikipedia provides a good reference. Also, the script will merge with any existing SQLite database with the same name, so take care to remove any previously existing database files before running the script.

On Planner Arena, switch to the "Change Database" tab, and upload your generated database. You should now be visualizing the results of your benchmarking.

Play around with the visualization, and view the various attributes of the planners. In particular, you will have generated results for the planners you previously used:

1. PRM: With the default settings.
2. RRT: With a range of 0.5, 5, and 25.
3. RRT-Connect: With a range of 0.5, 5, and 25.
4. KPIECE: With the default settings.

Before you can make conclusions from your benchmarking data, you must verify that the data you have

is *meaningful*. If a planner fails to solve a problem, values in the benchmarking results can range from informative to meaningless, as it is the product of an incomplete run of a planner. To verify that your benchmarking results have solved the problem, you should inspect the following two properties:

1. First, you should observe the "solved" attribute. A "1" corresponds to successfully solving the planning problem. A "0" corresponds to failing to solve the problem. Your planners should have mostly (more than 90%) successes.
2. Next, you should observe the "approximate solution" attribute. A "1" corresponds to only approximately solving the problem, but not successfully reaching the goal. A "0" corresponds to exactly solving the problem, reaching the goal. Your planners should mostly (more than 90%) have exact solutions, indicating they all solved the problem.

If your benchmarking results contain mostly failures, you will need to increase the allowed planning time of the benchmark by editing the `benchmarking` program parameters, and rerunning the benchmark.

Finally, observe how each of these planners performs on the following metrics:

1. "time": how much time was spent solving the problem?
2. "graph states": how many states (roughly how much memory) are in the planner's search graph?
3. "solution length": how long is the solution path?

2.5. Repeat the above procedures for robot `fetch`, scene `table_pick`.

```
./benchmarking --problem=table_pick --robot=fetch
```

Caution: The script `ompl_benchmark_statistics.py` will merge with any existing SQLite database with the same name, so take care to remove or rename any previously existing database files before running the script. Otherwise, **you may lose your data** from the previous benchmark.

2.6. At this point we will not ask you to run more benchmarking. We have run benchmarking on three other motion planning problems for you so that you do not need to run them; just interpret the results. The problems are called `Home`, `Twistycool`, and `Abstract`, which are all 3-D rigid body planning problems. For example, in `Home`, the problem is to find a motion plan for a free-flying table in a 3-D home environment, which resembles the famous Piano Movers problem. Below (towards the end of this handout), we provide screenshots of the three motion planning problems with their benchmarking results. You only need to analyze these data in your report. Figure **??** shows what the `Home` environment looks like, what an example solution path looks like, and the plots of the benchmarking data over this problem. Figure **??** and Figure **??** show the `Twistycool`, and `Abstract` environments respectively.

Deliverables

Submit a report, 3-5 pages in PDF format, that addresses the following:

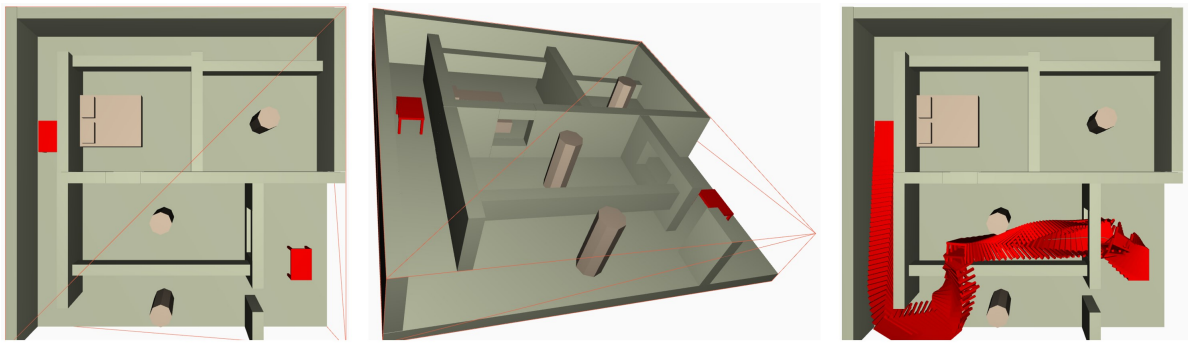
1. **(5 points)** In exercise **1.1**, in `RigidBodyPlanning.cpp`, we provide you with two ways to construct a motion planning problem. From your inspection, briefly explain 1) the common steps and 2) the differences between the two ways.
2. **(5 points)** Inspect `RigidBodyPlanningWithControls.cpp`; what are the differences between it and the simple setup planning routine from the previous program? Run each of them 5 times. On average which runs faster, and what do you think is the reason?
3. **(10 points)** Describe the perceived difficulty of each problem you tried in **1.2**. Please clearly state that in part **1.2**, which problem instance of each scene did you try (environment name and problem index)? Please clearly state on what metric you use to evaluate difficulty. According to your metric, which problems seemed easier and/or harder to solve than others?
4. **(10 points)** Out of the specified planners you ran in **1.2**, did any seem to perform better than the others? In what problem configurations did the planner perform better? On what metric are you evaluating planner performance? Elaborate.
5. **(20 points)** You should have noticed that each planner has a set of parameters. For PRM, there is "max nearest neighbors." For RRT, RRT-Connect, and KPIECE, there is "range" and "goal bias." Try varying the values of these parameters:

   (a) "max nearest neighbors": from 8 to 100
   (b) "range": from 0.5 to 25. Note that if 0, OMPL attempts to guess what a "good" value for the range should be.
   (c) "goal bias": from 0.01 to 0.95

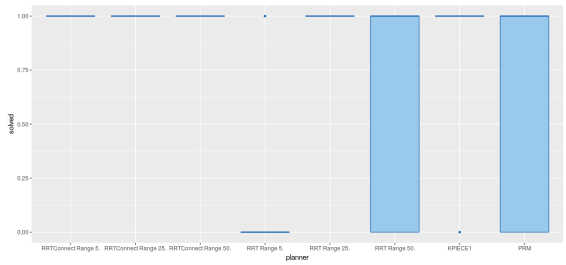
   Did you encounter any cases where changing a planner's parameters improved performance on a particular configuration in the first exercise? Explain.
6. **(45 points)** In exercise **2**, observe the benchmarking data for the five environments:

   - the three environments (`Home`, `Twistycool`, and `Abstract`) that we provide to you,
   - and the two environments you ran (`ur5` with `bookshelf_small` and `fetch` with `table_pick`) from the second exercise on Planner Arena.

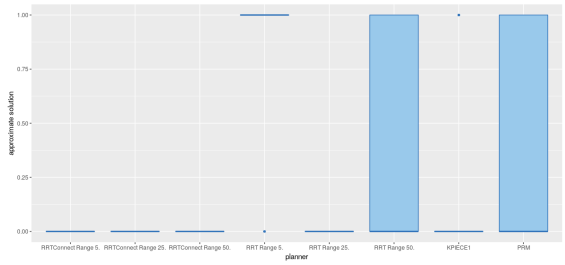
   Which planners perform best in which environments? Explain why, and by what metrics. Similar to the plots we provided to you, attach your own plots for the two that you ran to the report as figures that support your claims. Claims made without supporting evidence will be penalized.
7. **(5 points)** Rate the difficulty of each exercise on a scale of 1–10 (1 being trivial, 10 being impossible). Give an estimate of how many hours you spent on each exercise, and detail what was the hardest part of the assignment.



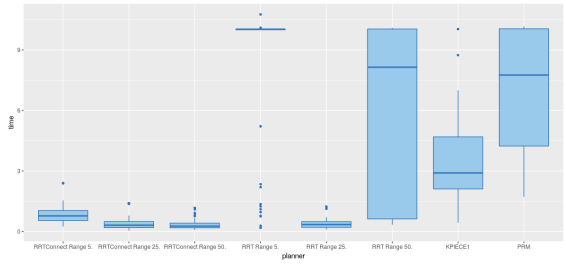
(a)



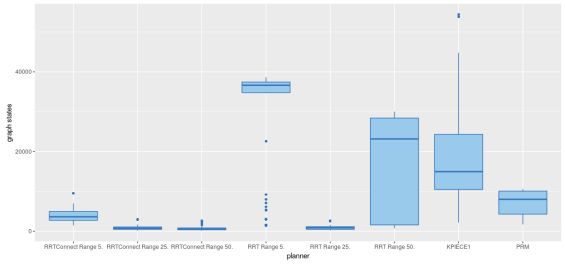
(b)



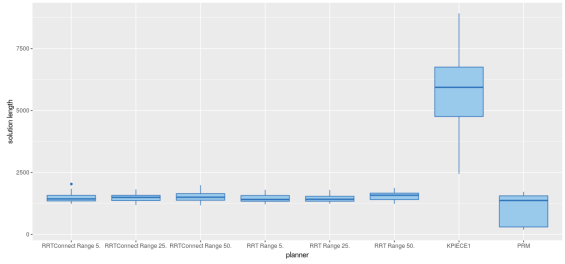
(c)



(d)

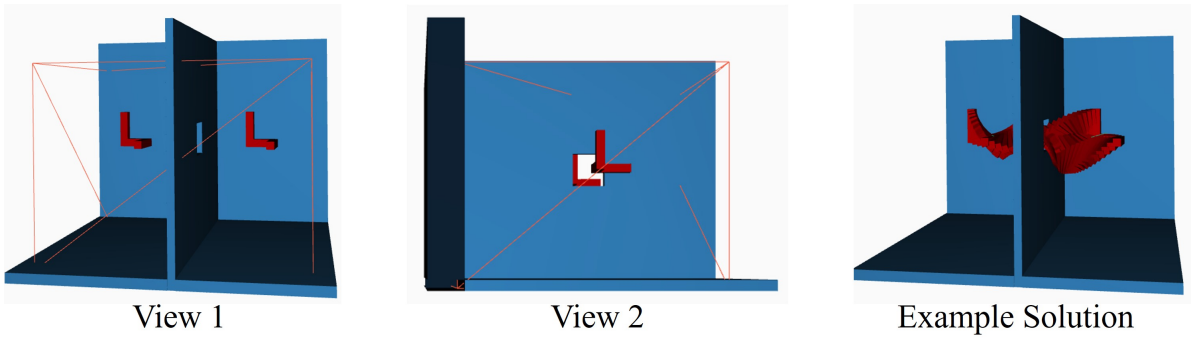


(e)

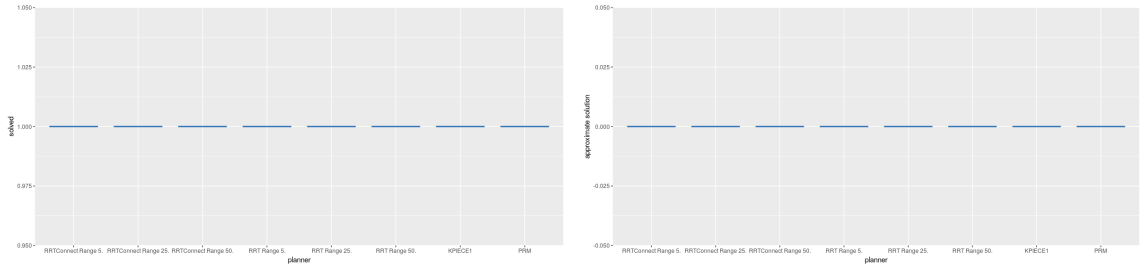


(f)

Figure 1: (a) The Home environment with two views and an example motion plan solution. (b) The "solved" attribute. (c) The "approximate solution" attribute. (d) The "time" attribute. (e) The "graph states" attribute. (f) The "path length" attribute. See section **2.4** for a detailed explanation of these attributes.

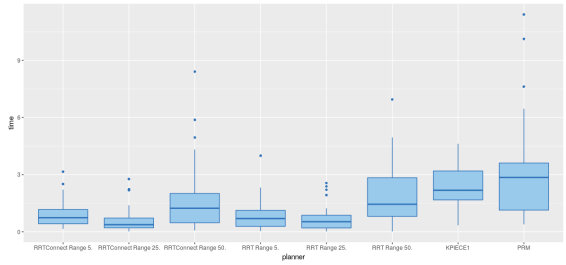


(a)

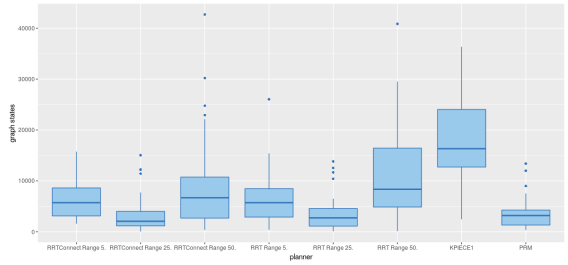


(b)

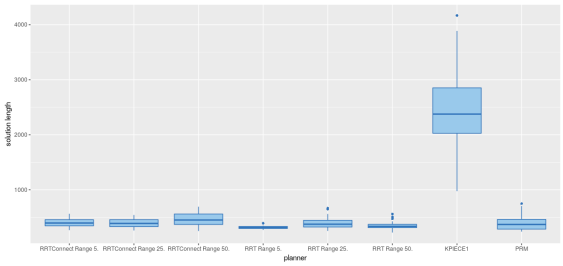
(c)



(d)

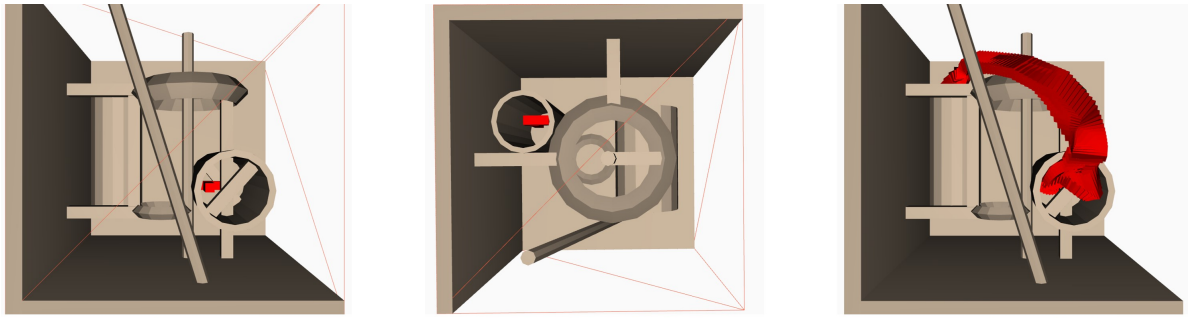


(e)

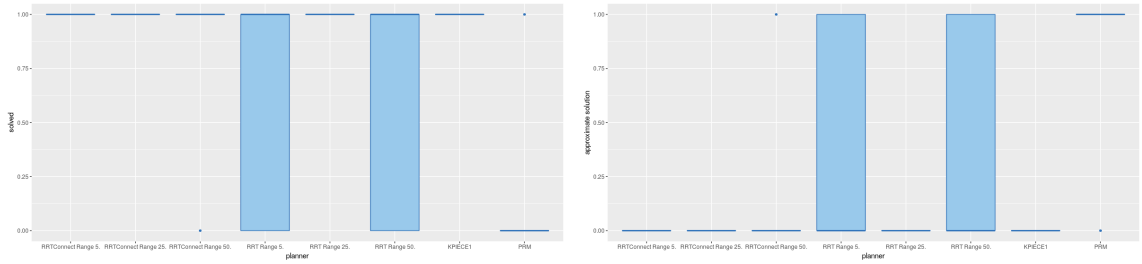


(f)

Figure 2: (a) The Twistycool environment with two views and an example motion plan solution. (b) The "solved" attribute. (c) The "approximate solution" attribute. (d) The "time" attribute. (e) The "graph states" attribute. (f) The "path length" attribute. See section **2.4** for detailed explanation of these attributes.

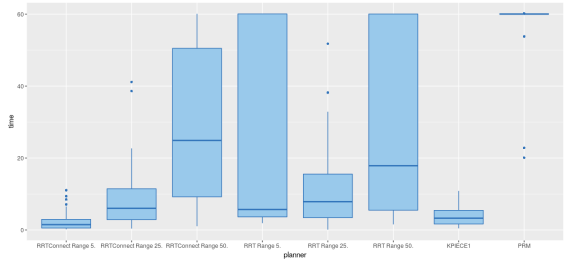


(a)

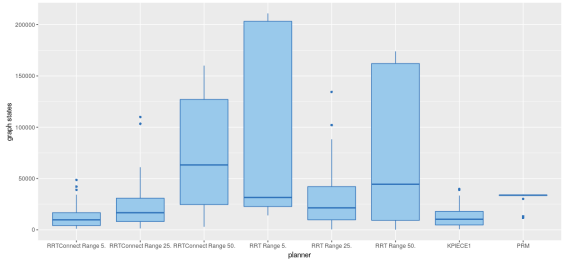


(b)

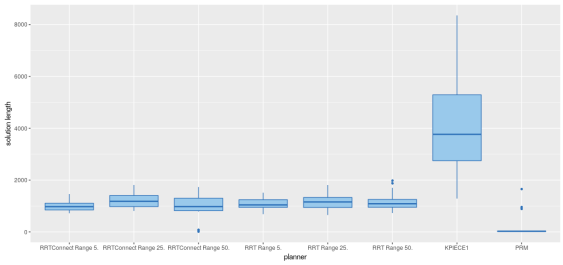
(c)



(d)



(e)



(f)

Figure 3: (a) The Abstract environment with two views and an example motion plan solution. (b) The “solved” attribute. (c) The “approximate solution” attribute. (d) The “time” attribute. (e) The “graph states” attribute. (f) The “path length” attribute. See section **2.4** for detailed explanation of these attributes.