Path Planning (Advanced)

Jay Patrikar and Brady Moon

7 July 2020

Carnegie Mellon University
The Robotics Institute

air lab

Session Objectives

- Show the story of development and progression of motion planning algorithms
- Overview of a few of the newer algorithms and their advantages
- Introduce you to the easy-to-use Open Motion Planning Library (OMPL) to quickly implement a myriad of algorithms
- Give you hands-on experience with OMPL

Outline

- Review
- FMT*
- Informed RRT*
- BIT*
- RABIT*
- OMPL
- Exercises

Previously Covered

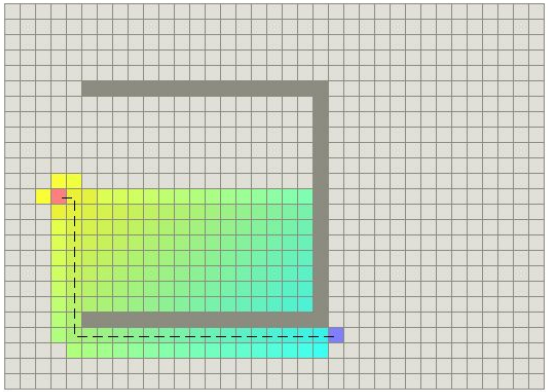
- Reviewed the planning problem for autonomous flying robots
- Abstraction and Approach
- Considered several algorithms and the idea of planning ensembles
- Which approaches are successful is highly dependent on your environment and dynamical system

Previously Covered

- 3 Representative approaches
  - Regular graph search: A*-grid search
  - Sampling-based: RRT*
  - Trajectory optimization: CHOMP

Review: Graph-search methods

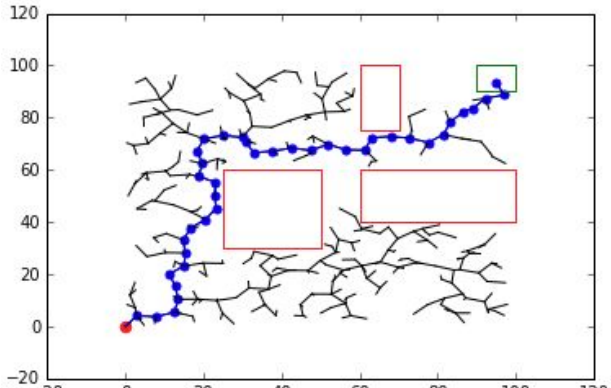
- Can use dynamic programming
- Find an exact solution to a discrete approximation of the problem
- Resolution complete
- Resolution optimal
- Optimally efficient (A*)
- Cons
  - Quality of the continuous path depends on the discretization of the problem
  - Suffers in high-dimensional spaces (curse of dimensionality)



[Amit]

Review: Sampling-based methods

- No discretization (sample from a continuous space)
- Scales well to higher dimensions
- Probabilistically complete
- Anytime resolution
- Can be asymptotically optimal (RRT*)
- Cons
  - Random sampling can lead to long computation times



[Lopez]

Review: Sampling-based methods

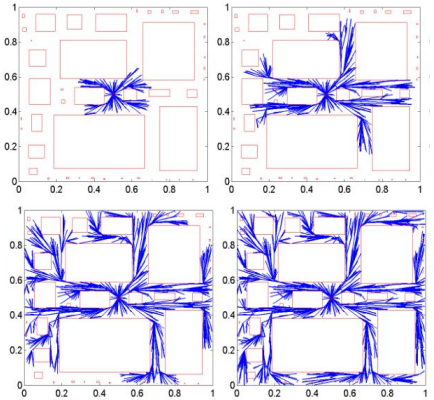
- Multiple-query and single-query.
  - Construct a graph (roadmap)
  - Can have multiple start and ending state pairs
  - Examples include PRM, Lazy-PRM, dynamic PRM, and PRM*
- Single-query
  - One start and end state
  - Examples include RRT, RDT, RRT*, EST, SRT, RRM.

What are the latest developments?

- Dijkstra's Algorithm 1956
- A* 1968
- RRT 1998
- RRT* 2010
- CHOMP 2013

FMT*

- *Fast Marching Tree: a Fast Marching Sampling-Based Method for Optimal Motion Planning in Many Dimensions** [Janson15]
- Combines features of multiple and single query algorithms
- Performs a lazy dynamic programming recursion on a set of samples
- Asymptotically optimal
- Faster convergence than RRT* and PRM*
- Incrementally builds out from starting state
- Never needs to backtrack over previously evaluated points



[Janson15]

FMT* Advantages

- Better than RRT* because FMT* creates connections near optimally.
- No rewiring
- Better than PRM* because FMT* builds paths in a tree-like structure which is good when differential constraints. Works outward so knows optimal path. Just adding to the end.
- When no obstacles, same solution as PRM* but faster computation
- Works best when high density of obstacles

FMT* Overview

Algorithm 1 Fast Marching Tree Algorithm (FMT*): Basics

Require: sample set V comprising of x_{init} and n samples in $\mathcal{X}_{\text{free}}$, at least one of which is also in $\mathcal{X}_{\text{goal}}$

Place x_{init} in V_{open} and all other samples in $V_{\text{unvisited}}$; initialize tree with root node x_{init}
Find lowest-cost node z in V_{open}
    For each of z's neighbors x in $V_{\text{unvisited}}$:
        Find neighbor nodes y in V_{open}
        Find locally-optimal one-step connection to x from among nodes y
        If that connection is collision-free, add edge to tree of paths
    Remove successfully connected nodes x from $V_{\text{unvisited}}$ and add them to V_{open}
    Remove z from V_{open} and add it to V_{closed}
    Repeat until either:
        (1) V_{open} is empty $\Rightarrow$ report failure
        (2) Lowest-cost node z in V_{open} is in $\mathcal{X}_{\text{goal}} \Rightarrow$ return unique path to z and report success

[Janson15]

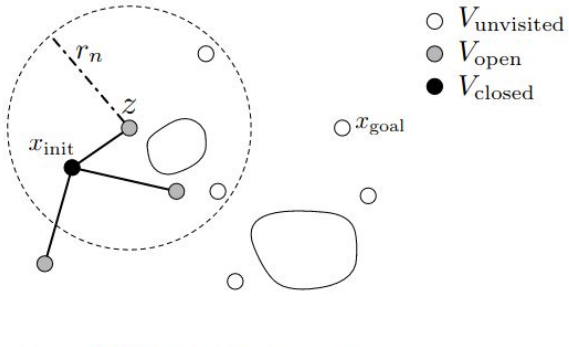
FMT* Overview

Algorithm 1 Fast Marching Tree Algorithm (FMT*): Basics

Require: sample set V comprising of x_{init} and n samples in $\mathcal{X}_{\text{free}}$, at least one of which is also in $\mathcal{X}_{\text{goal}}$

Place x_{init} in V_{open} and all other samples in $V_{\text{unvisited}}$; initialize tree with root node x_{init}
Find lowest-cost node z in V_{open}
    For each of z's neighbors x in $V_{\text{unvisited}}$:
        Find neighbor nodes y in V_{open}
        Find locally-optimal one-step connection to x from among nodes y
        If that connection is collision-free, add edge to tree of paths
    Remove successfully connected nodes x from $V_{\text{unvisited}}$ and add them to V_{open}
    Remove z from V_{open} and add it to V_{closed}
    Repeat until either:
        (1) V_{open} is empty $\Rightarrow$ report failure
        (2) Lowest-cost node z in V_{open} is in $\mathcal{X}_{\text{goal}} \Rightarrow$ return unique path to z and report success

[Janson15]



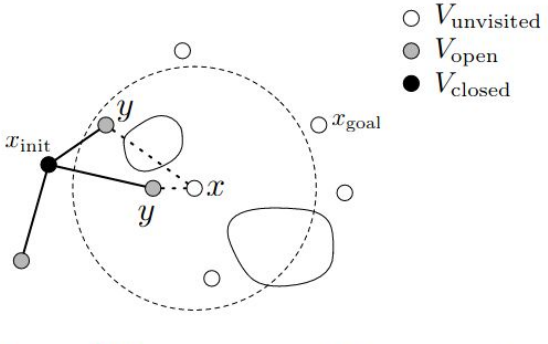
FMT* Overview

Algorithm 1 Fast Marching Tree Algorithm (FMT*): Basics

Require: sample set V comprising of x_{init} and n samples in $\mathcal{X}_{\text{free}}$, at least one of which is also in $\mathcal{X}_{\text{goal}}$

1: Place x_{init} in V_{open} and all other samples in $V_{\text{unvisited}}$; initialize tree with root node x_{init}
2: Find lowest-cost node z in V_{open}
3: For each of z's neighbors x in $V_{\text{unvisited}}$:
4: Find neighbor nodes y in V_{open}
5: Find locally-optimal one-step connection to x from among nodes y
6: If that connection is collision-free, add edge to tree of paths
7: Remove successfully connected nodes x from $V_{\text{unvisited}}$ and add them to V_{open}
8: Remove z from V_{open} and add it to V_{closed}
9: Repeat until either:
(1) V_{open} is empty $\Rightarrow$ report failure
(2) Lowest-cost node z in V_{open} is in $\mathcal{X}_{\text{goal}} \Rightarrow$ return unique path to z and report success

[Janson15]



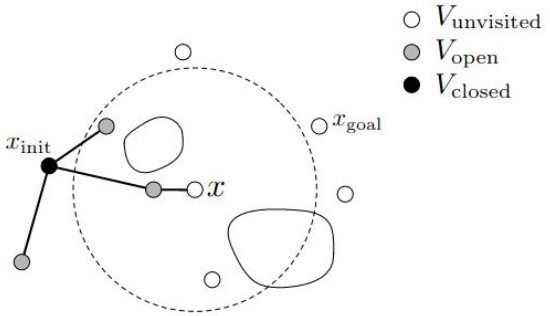
FMT* Overview

Algorithm 1 Fast Marching Tree Algorithm (FMT*): Basics

Require: sample set V comprising of x_{init} and n samples in $\mathcal{X}_{\text{free}}$, at least one of which is also in $\mathcal{X}_{\text{goal}}$

1: Place x_{init} in V_{open} and all other samples in $V_{\text{unvisited}}$; initialize tree with root node x_{init}
2: Find lowest-cost node z in V_{open}
3: For each of z's neighbors x in $V_{\text{unvisited}}$:
4: Find neighbor nodes y in V_{open}
5: Find locally-optimal one-step connection to x from among nodes y
6: If that connection is collision-free, add edge to tree of paths
7: Remove successfully connected nodes x from $V_{\text{unvisited}}$ and add them to V_{open}
8: Remove z from V_{open} and add it to V_{closed}
9: Repeat until either:
(1) V_{open} is empty $\Rightarrow$ report failure
(2) Lowest-cost node z in V_{open} is in $\mathcal{X}_{\text{goal}} \Rightarrow$ return unique path to z and report success

[Janson15]



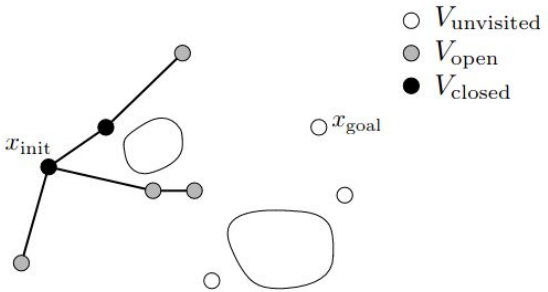
FMT* Overview

Algorithm 1 Fast Marching Tree Algorithm (FMT*): Basics

Require: sample set V comprising of x_{init} and n samples in $\mathcal{X}_{\text{free}}$, at least one of which is also in $\mathcal{X}_{\text{goal}}$

1: Place x_{init} in V_{open} and all other samples in $V_{\text{unvisited}}$; initialize tree with root node x_{init}
2: Find lowest-cost node z in V_{open}
3: For each of z's neighbors x in $V_{\text{unvisited}}$:
4: Find neighbor nodes y in V_{open}
5: Find locally-optimal one-step connection to x from among nodes y
6: If that connection is collision-free, add edge to tree of paths
7: Remove successfully connected nodes x from $V_{\text{unvisited}}$ and add them to V_{open}
8: Remove z from V_{open} and add it to V_{closed}
9: Repeat until either:
(1) V_{open} is empty $\Rightarrow$ report failure
(2) Lowest-cost node z in V_{open} is in $\mathcal{X}_{\text{goal}} \Rightarrow$ return unique path to z and report success

[Janson15]



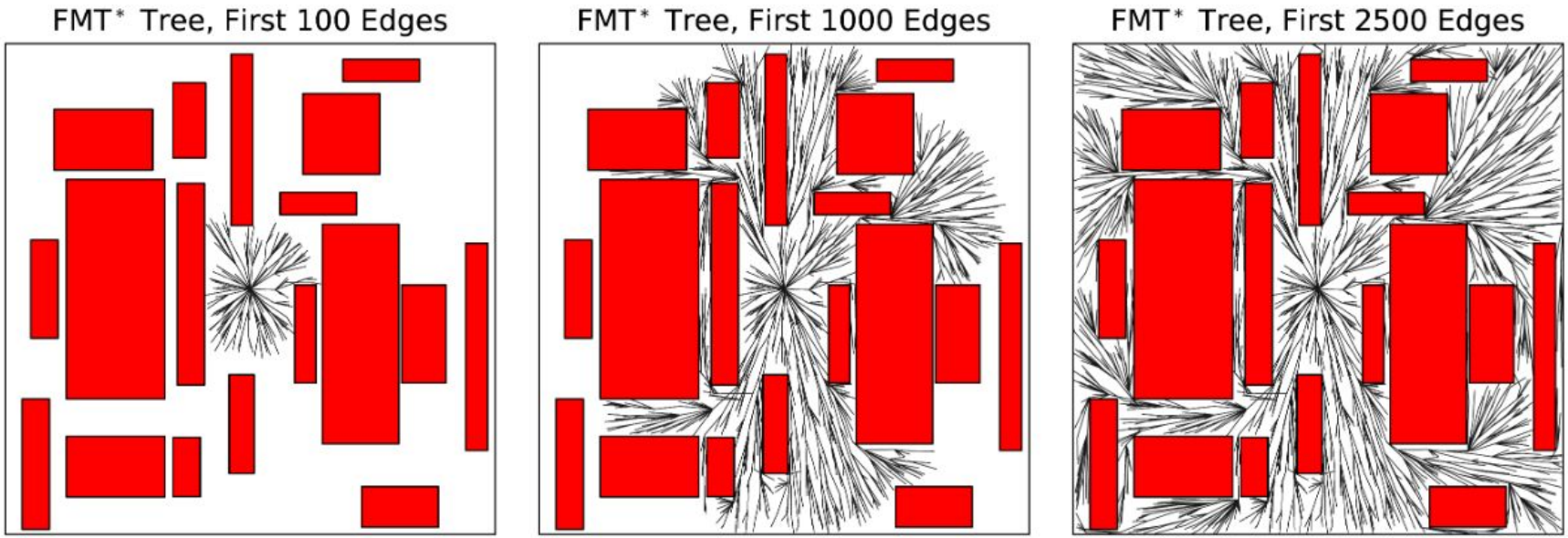
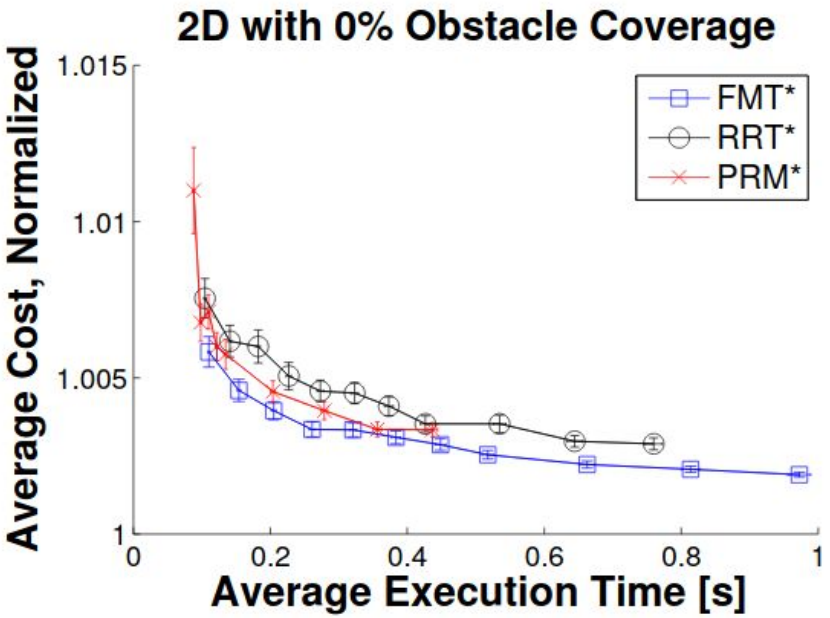
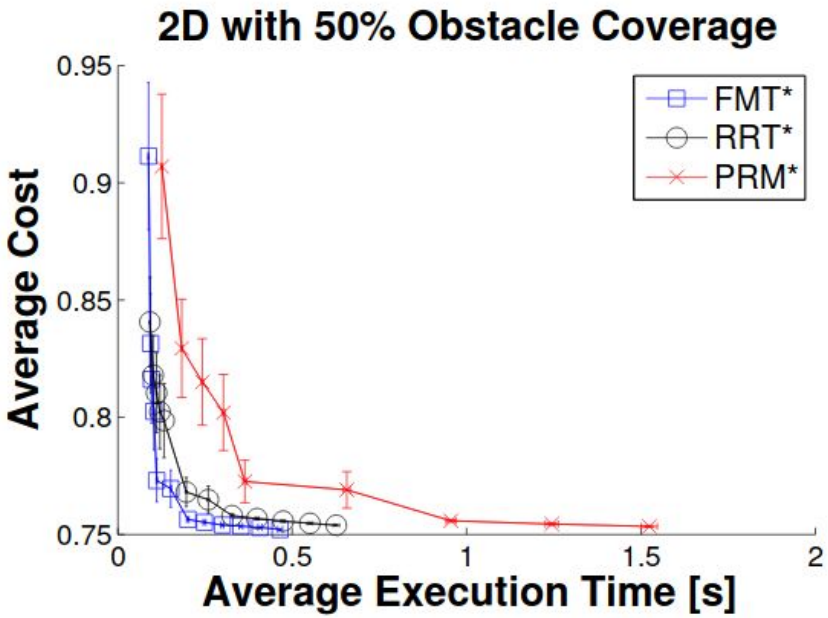


Figure 1: The FMT* algorithm generates a tree by moving steadily outward in cost-to-arrive space. This figure portrays the growth of the tree in a 2D environment with 2,500 samples (only edges are shown).

[Janson15]

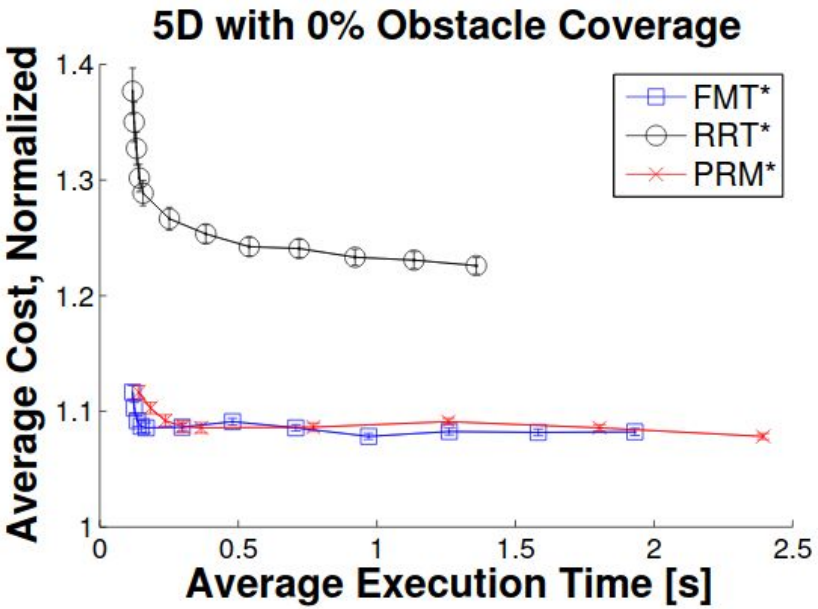
Results: 2D

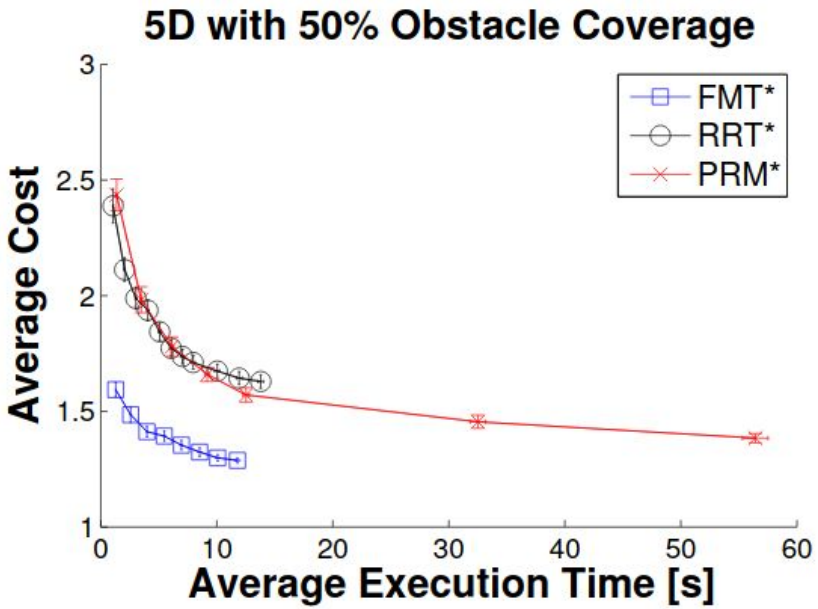




[Janson15]

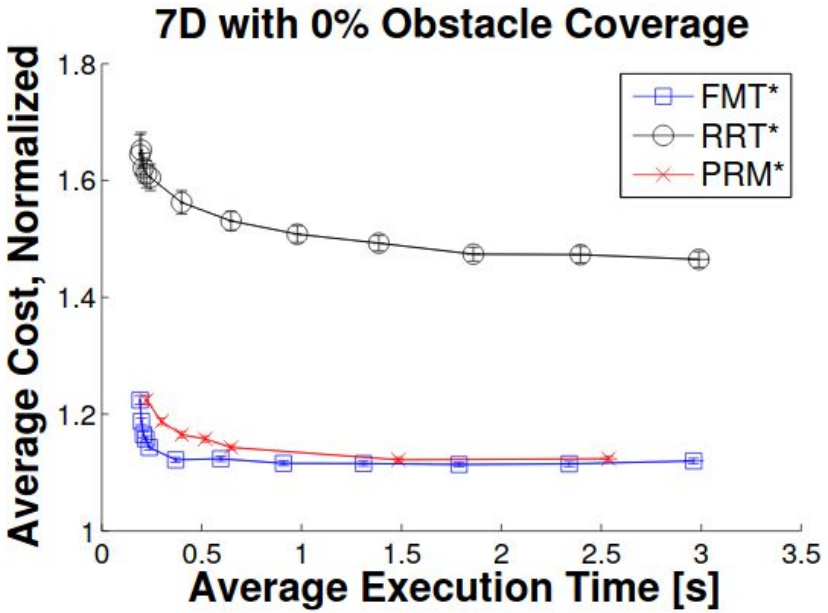
Results: 5D

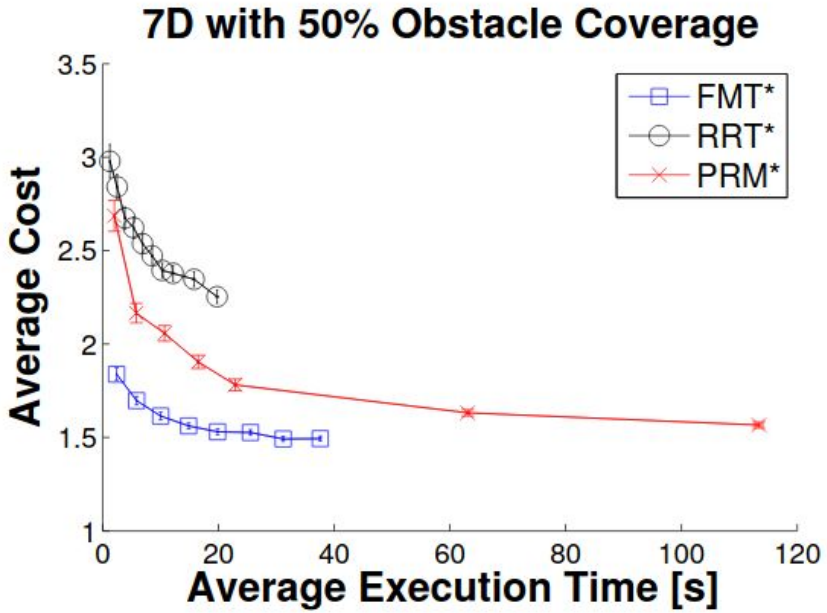




[Janson15]

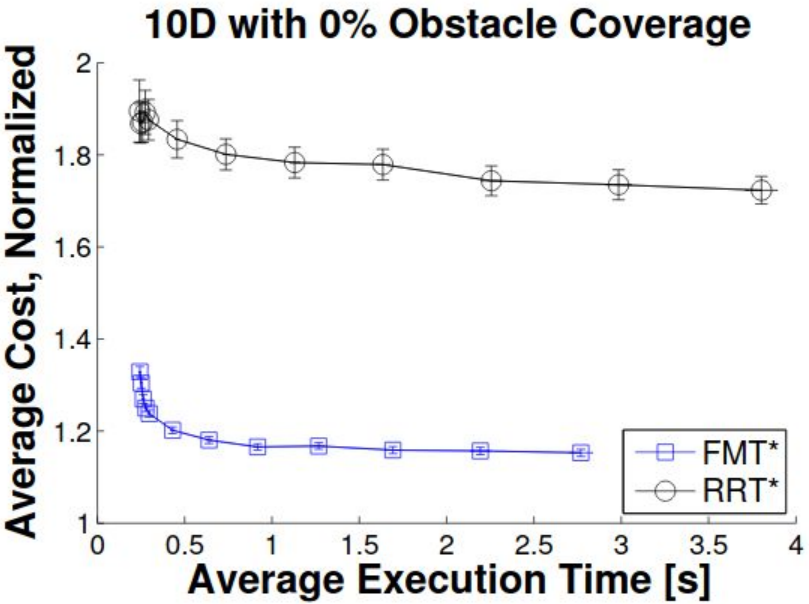
Results: 7D

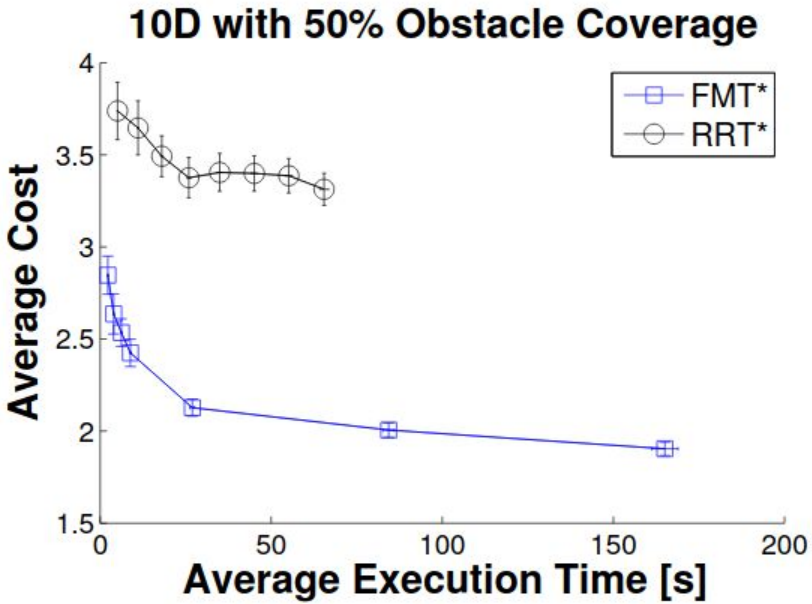




[Janson15]

Results: 10D

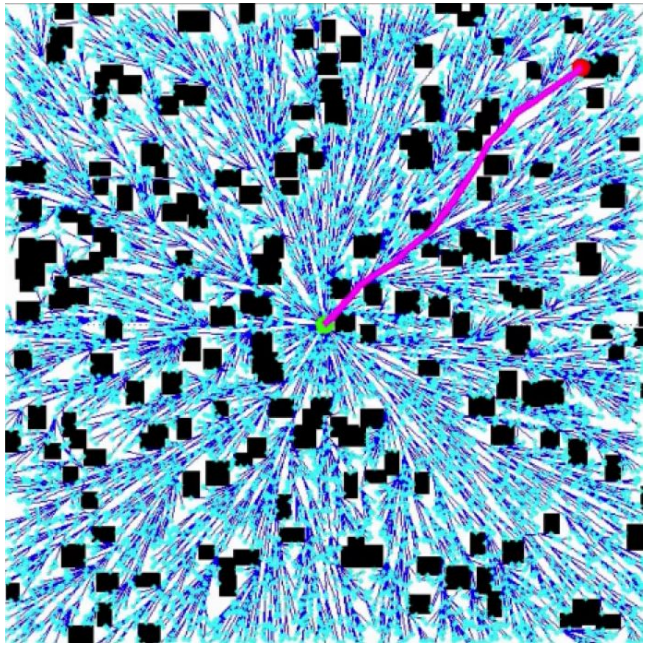




[Janson15]

What could be improved?

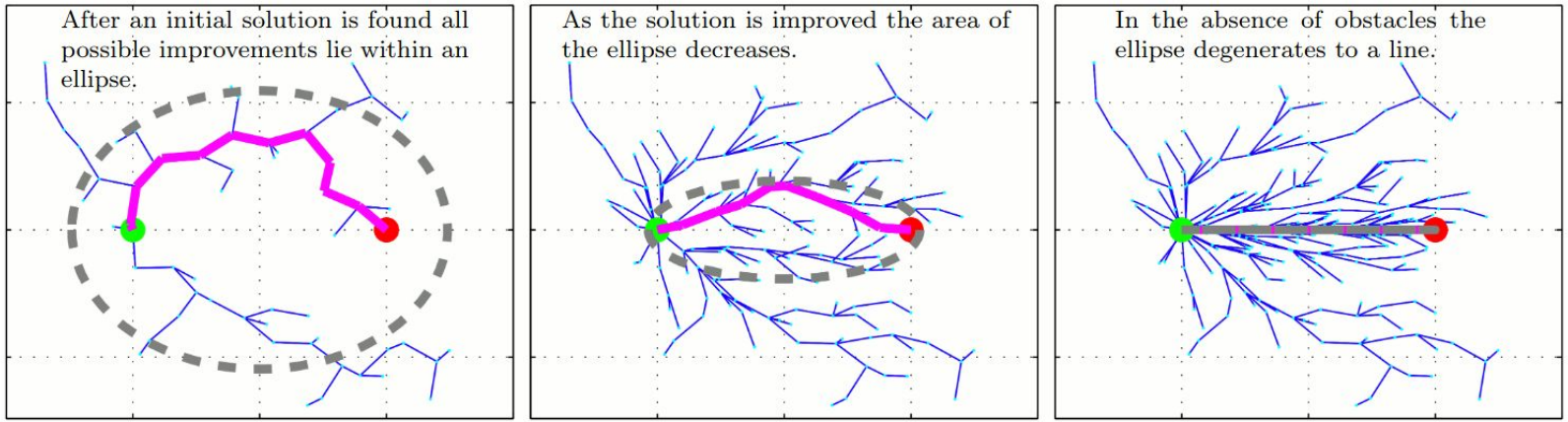
- Solving for optimal paths to every state is inefficient for many applications



[Gammell14]

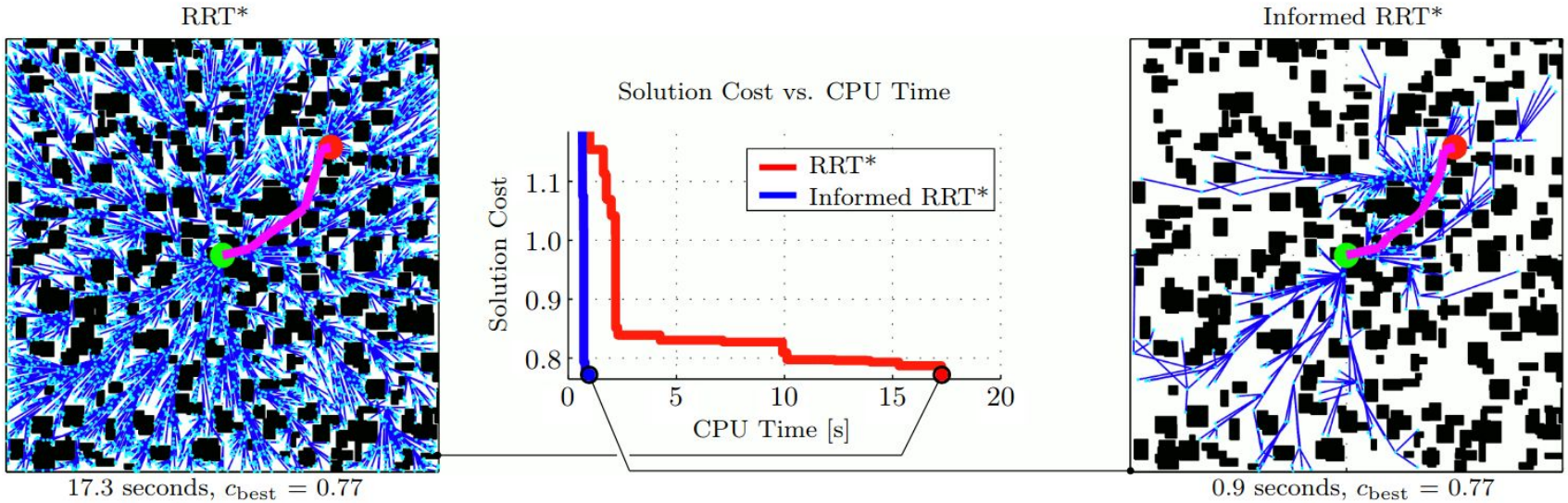
Informed RRT*

- Narrow search to a subproblem that would lead to a better solution
- Subproblem defined by an n-dimensional ellipse



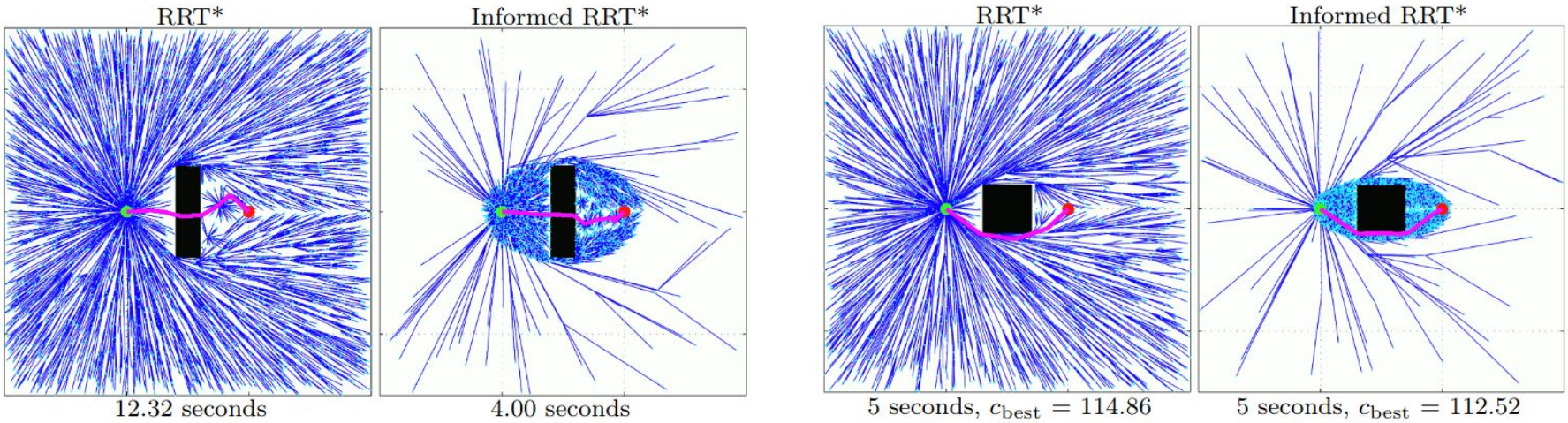
[Gammell14]

Informed RRT*



[Gammell14]

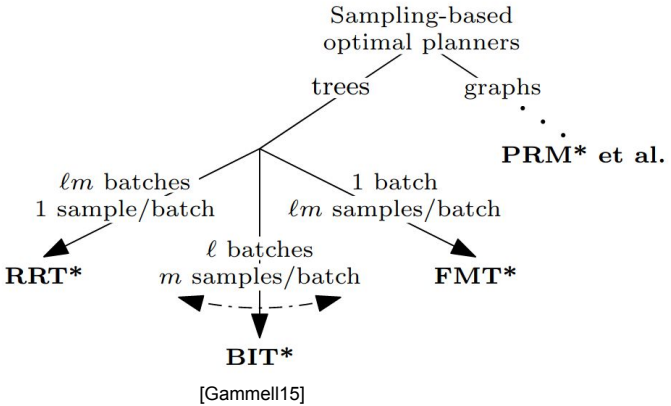
Informed RRT*



[Gammell14]

Can we get the best RRT* and FMT*?

- *Batch Informed Trees (BIT*): Sampling-based Optimal Planning via the Heuristically Guided Search of Implicit Random Geometric Graphs* (Gammel et al. 2015)
- Finds a balance between the benefits of graph-search and sampling-based methods
- Benefits of batches, but also has anytime resolution
- Limits search space (Informed RRT*)



[Gammell15]

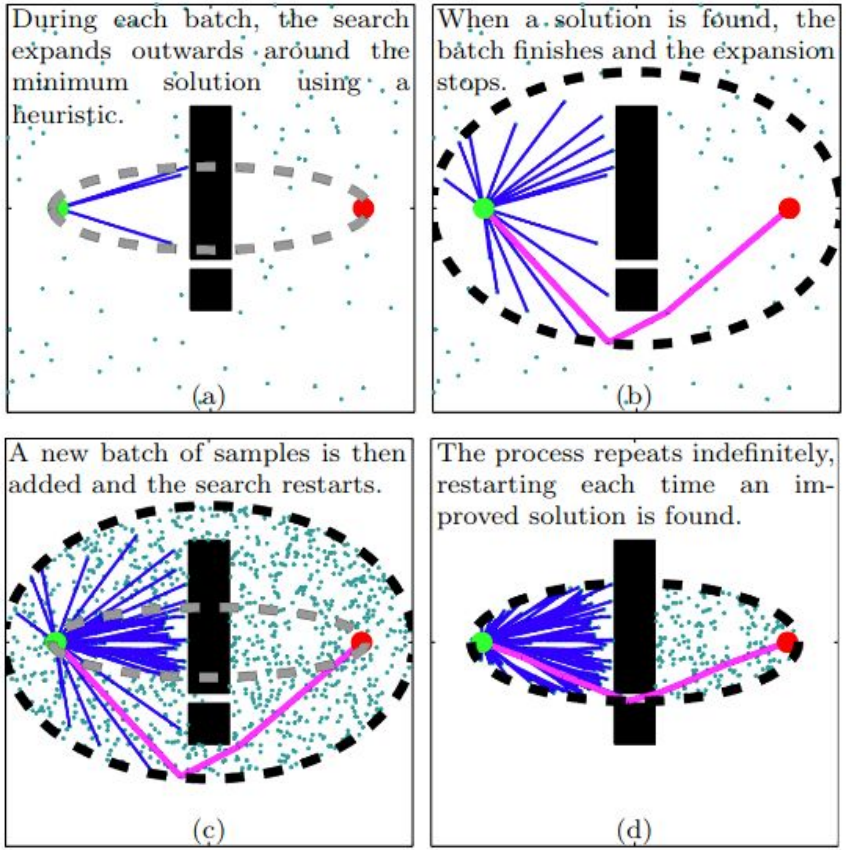
BIT*

- Uses batches of random samples
- Samples from an implicit random geometric graph (RGG)
- Uses a heuristic to search the RGG in order of decreasing solution quality

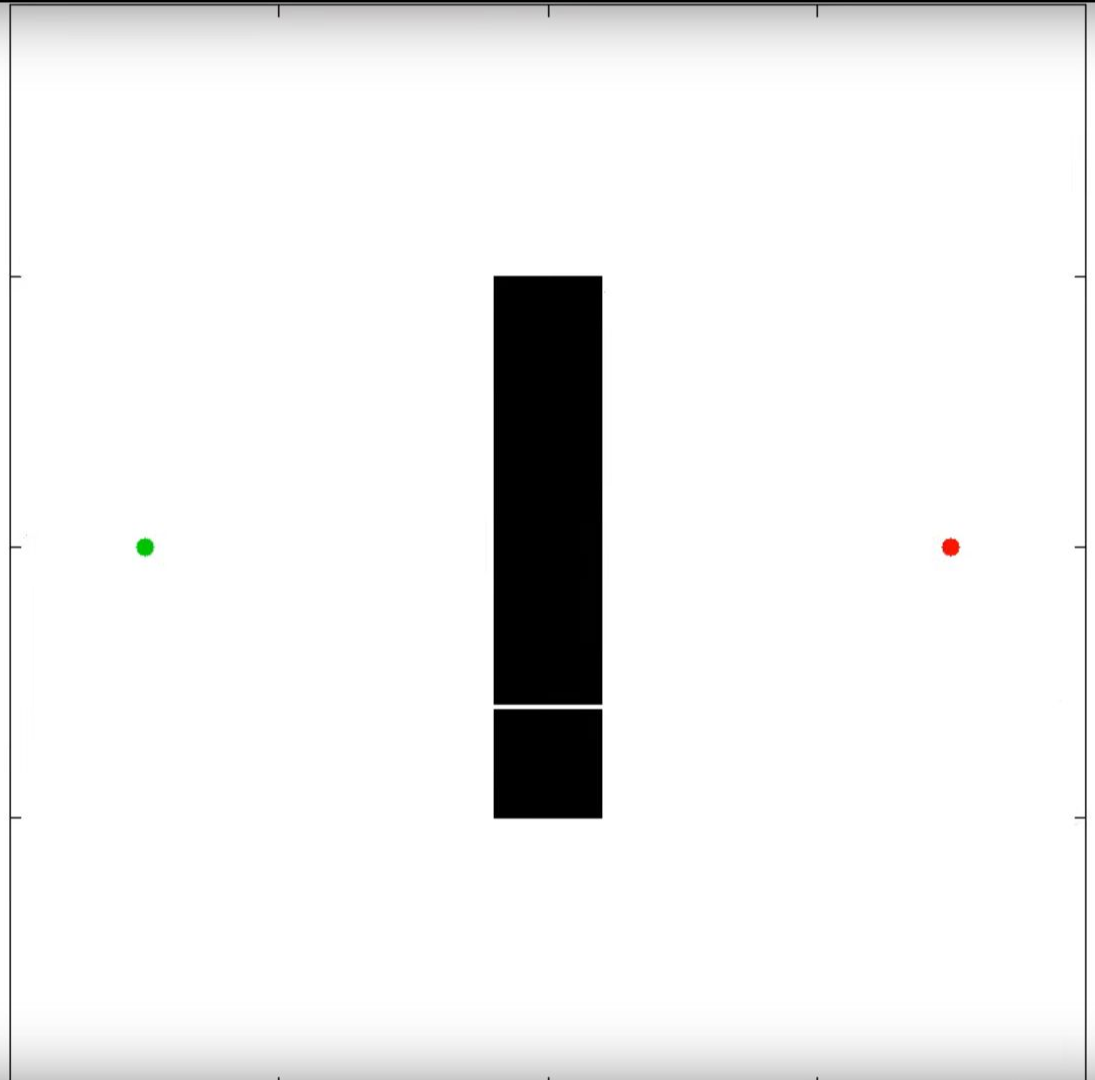
$$\texttt{radius}\,(q) := 2\eta\left(1+\tfrac{1}{n}\right)^{\frac{1}{n}}\left(\frac{\lambda\left(X_{\hat{f}}\right)}{\zeta_n}\right)^{\frac{1}{n}}\left(\frac{\log(q)}{q}\right)^{\frac{1}{n}}$$

[Gammell15]

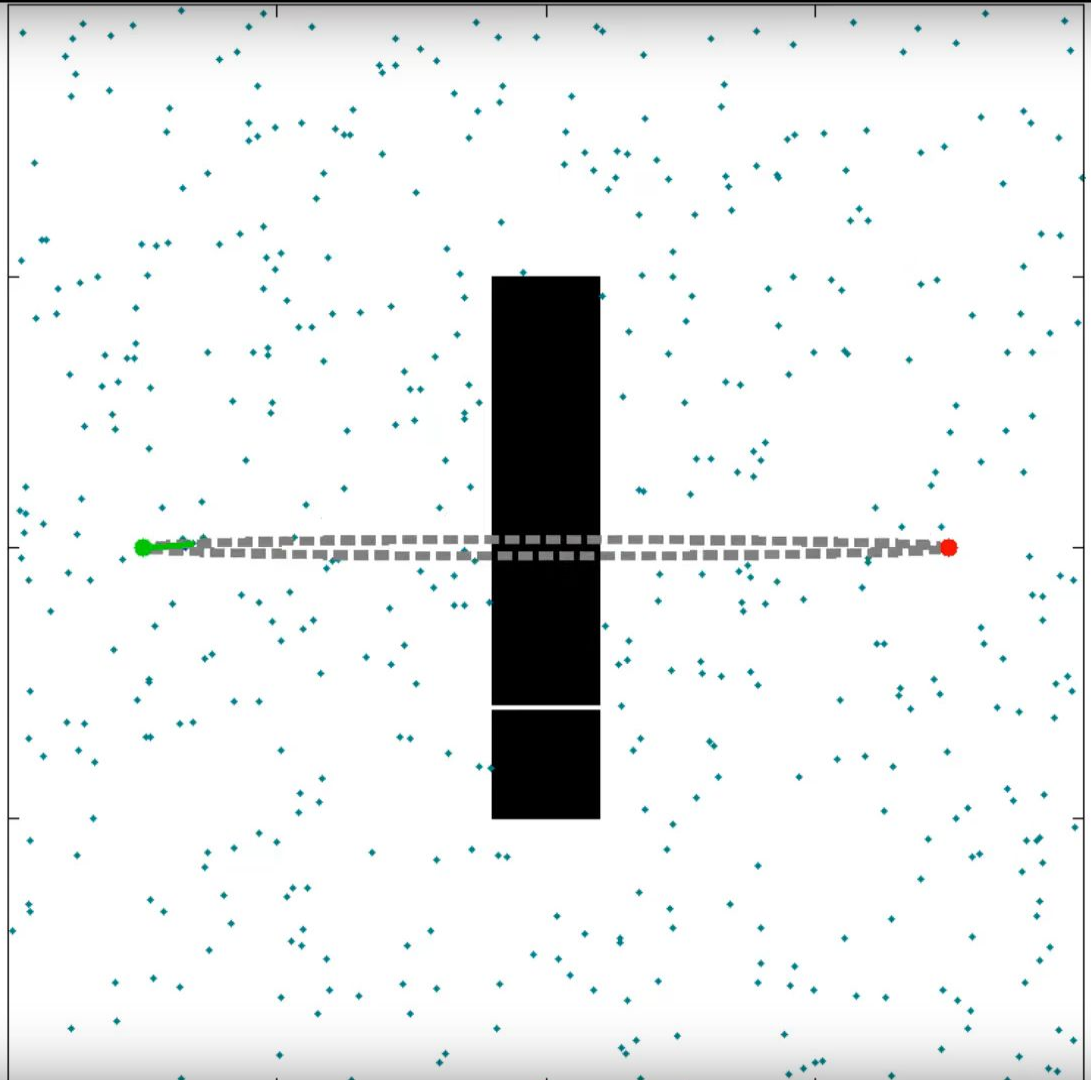
BIT*



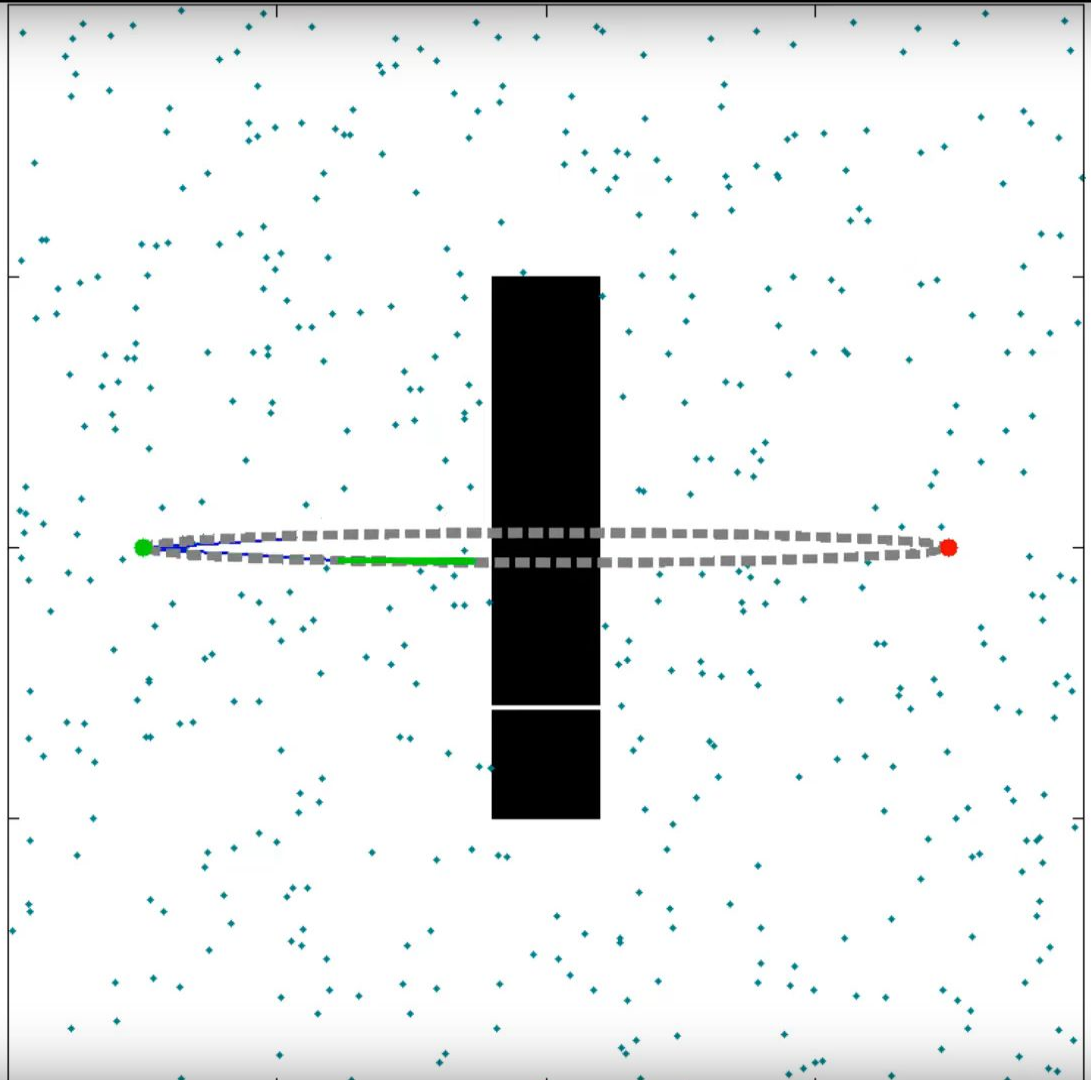
[Gammell15]



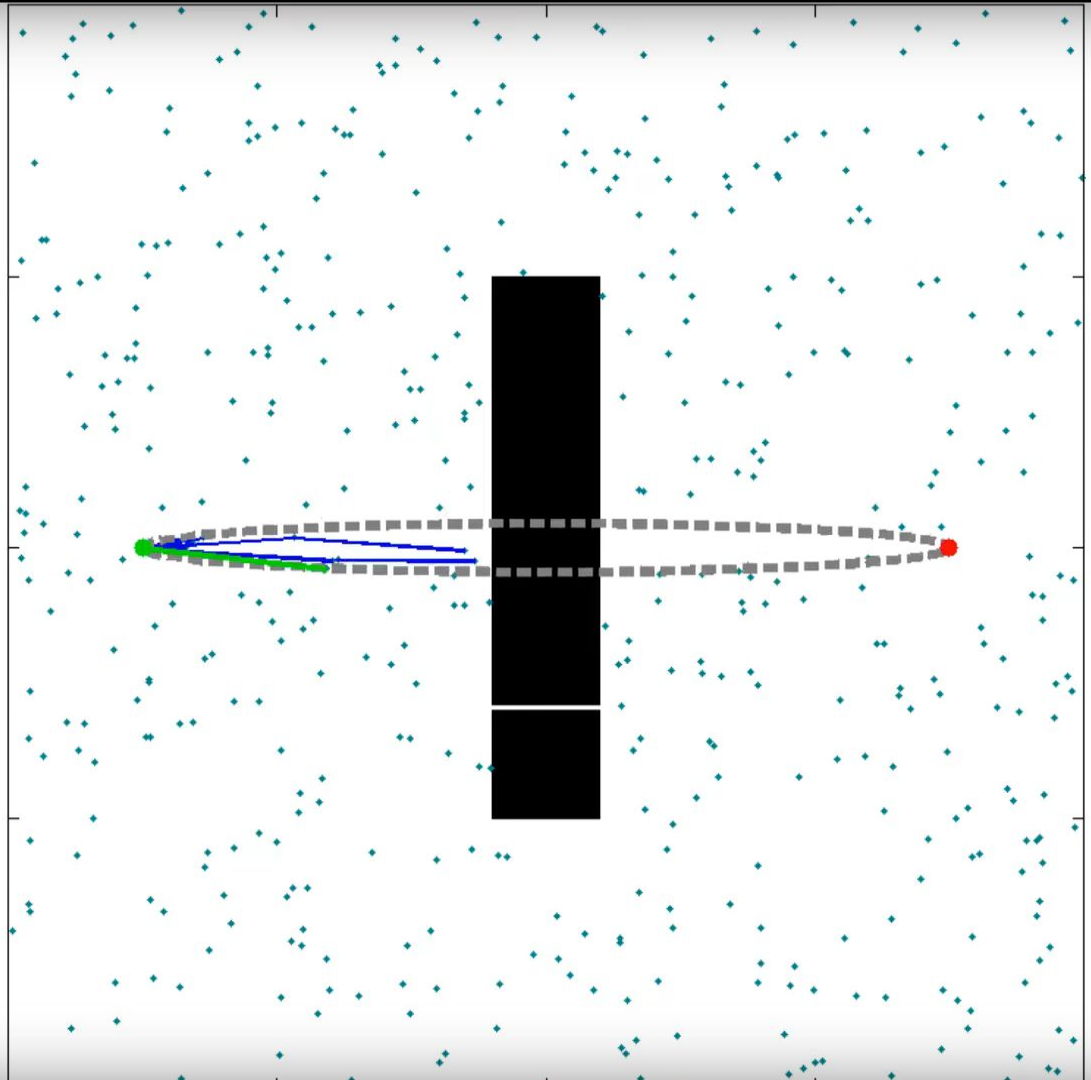
https://www.youtube.com/watch?v=TQIoCC48gp4



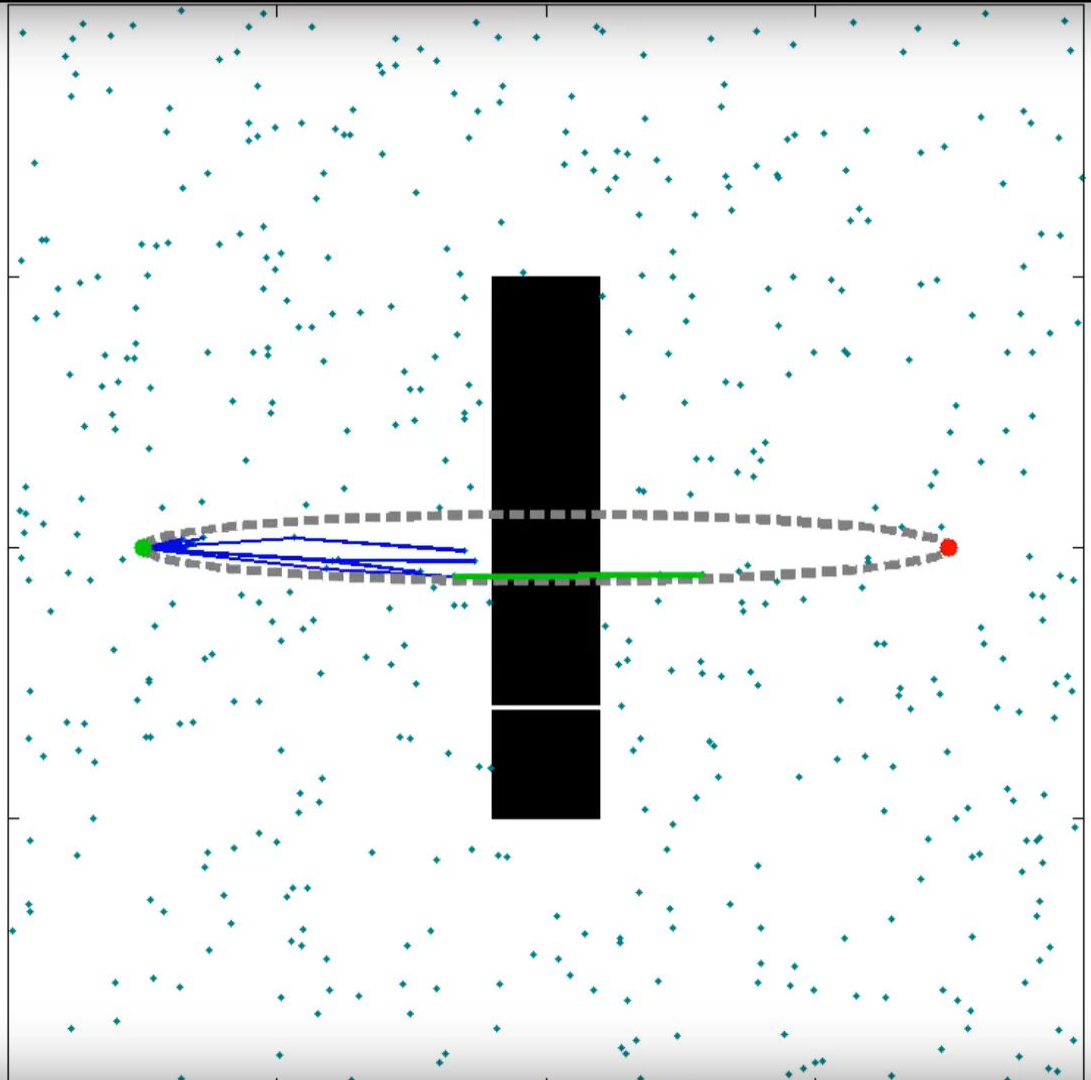
https://www.youtube.com/watch?v=TQIoCC48gp4



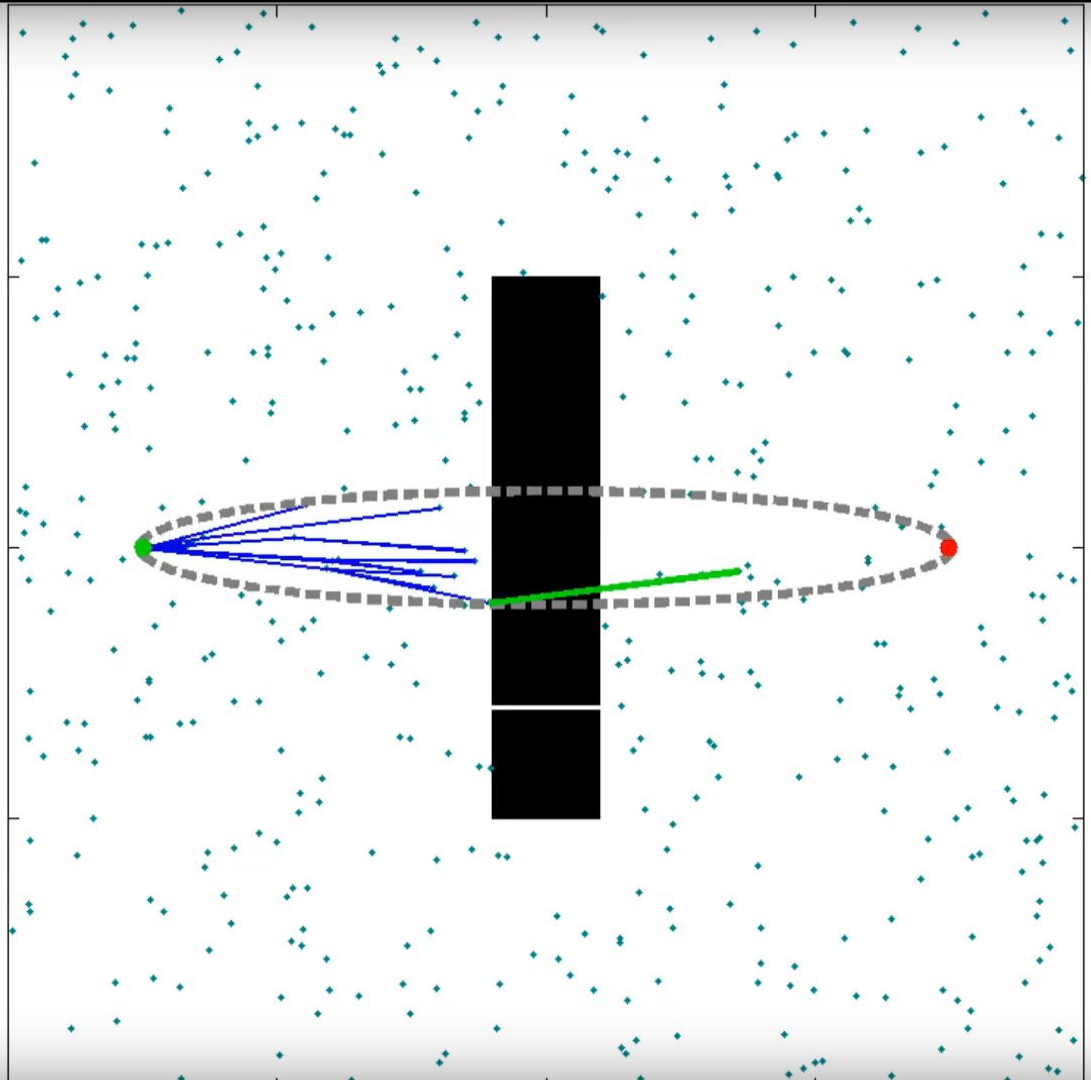
https://www.youtube.com/watch?v=TQIoCC48gp4



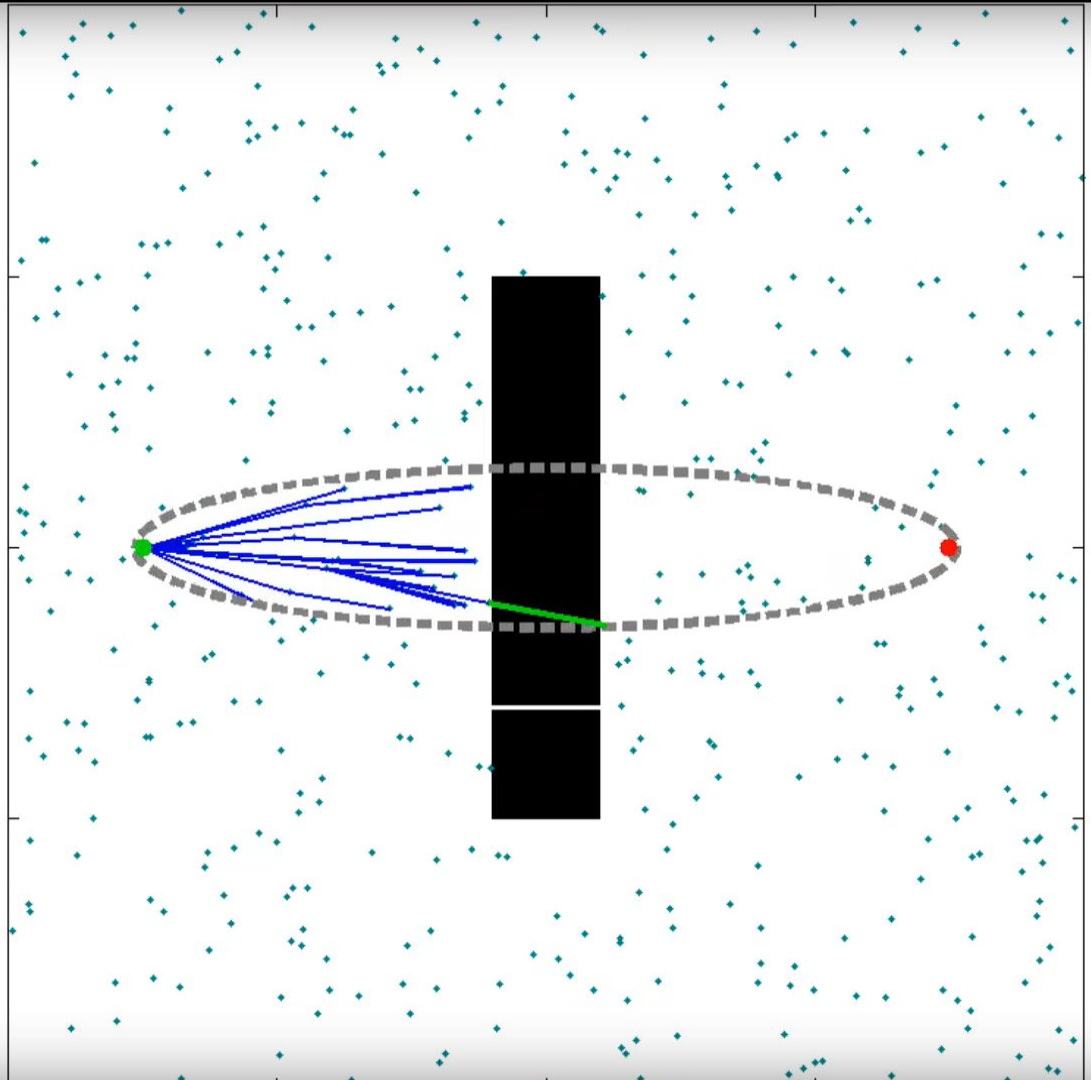
https://www.youtube.com/watch?v=TQIoCC48gp4



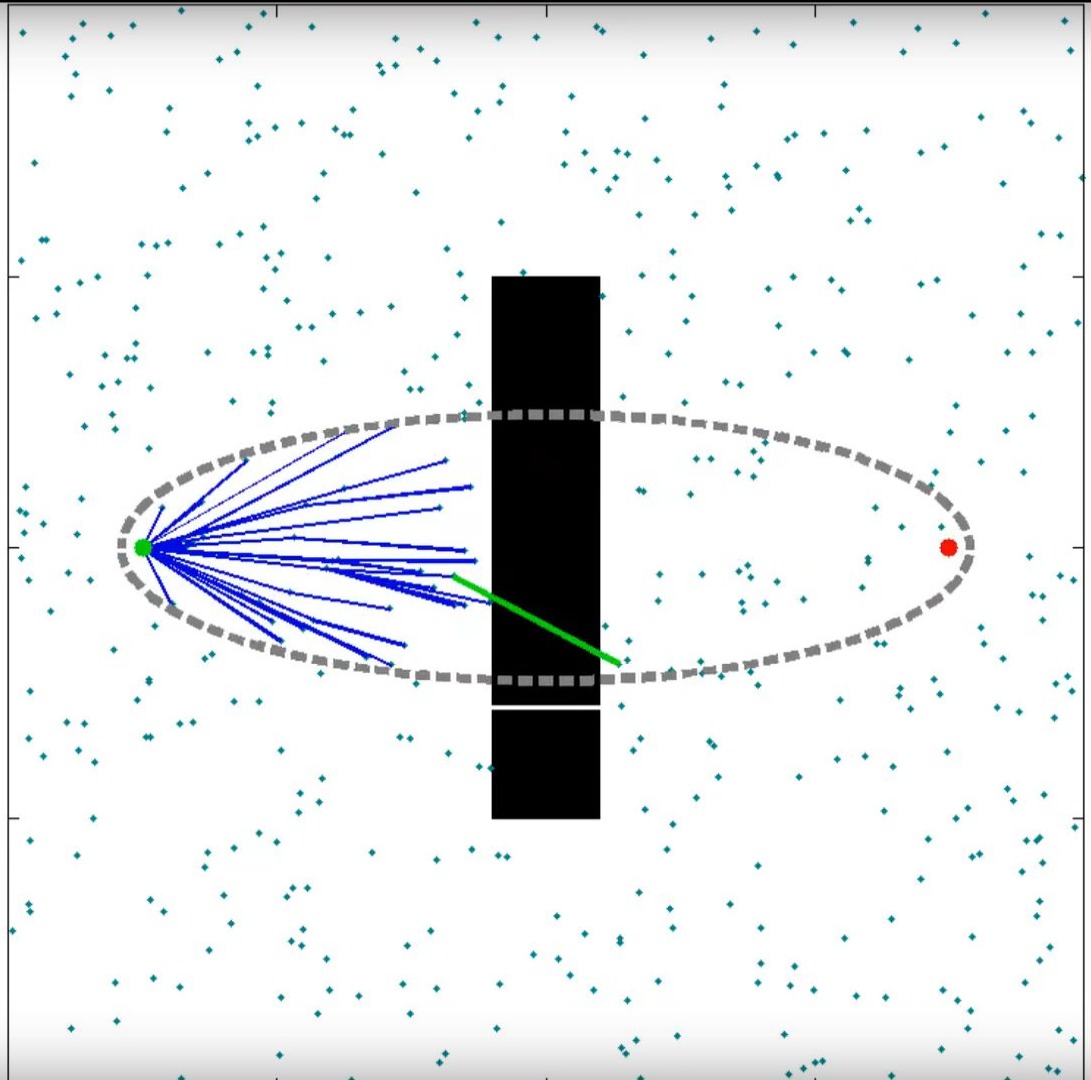
https://www.youtube.com/watch?v=TQIoCC48gp4



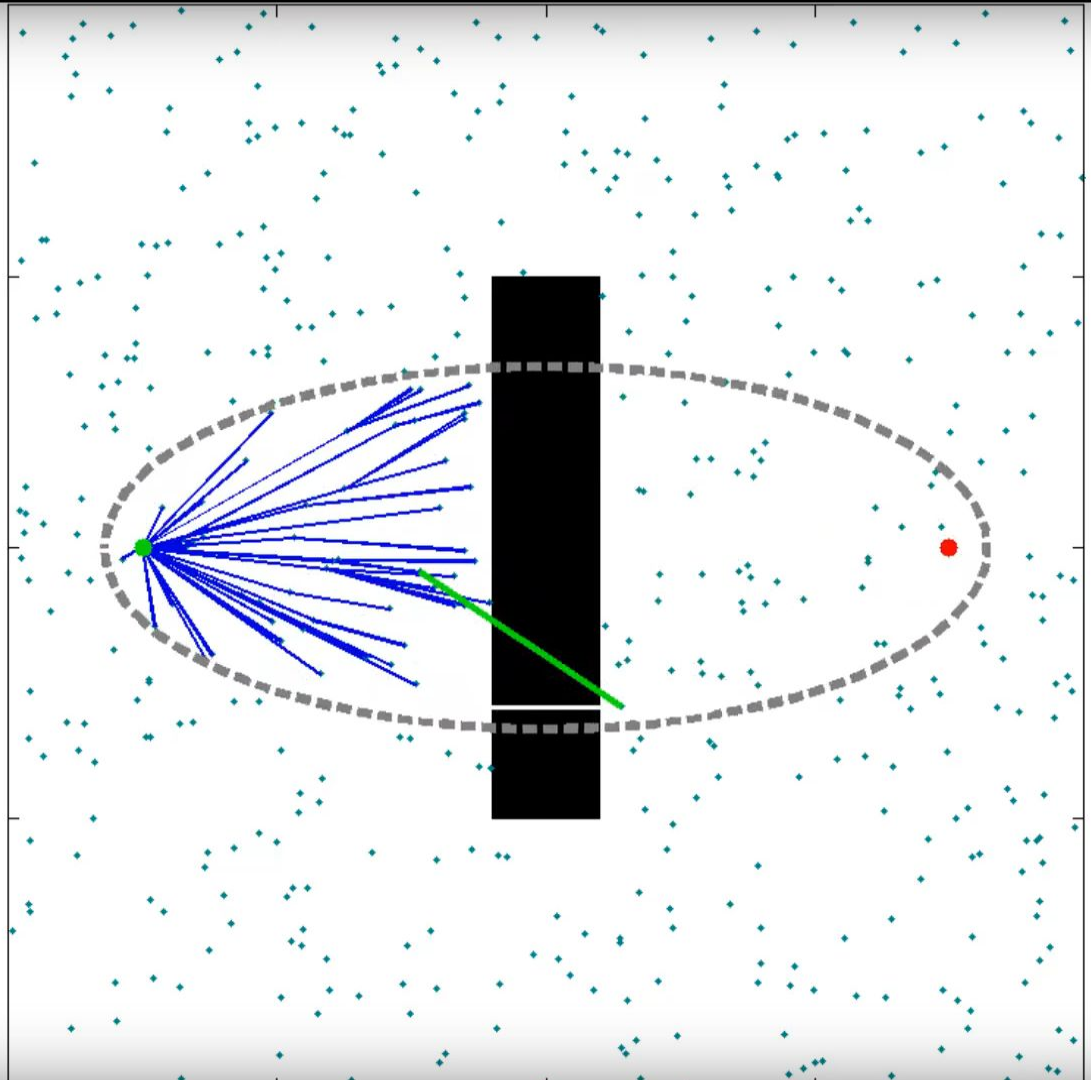
https://www.youtube.com/watch?v=TQIoCC48gp4



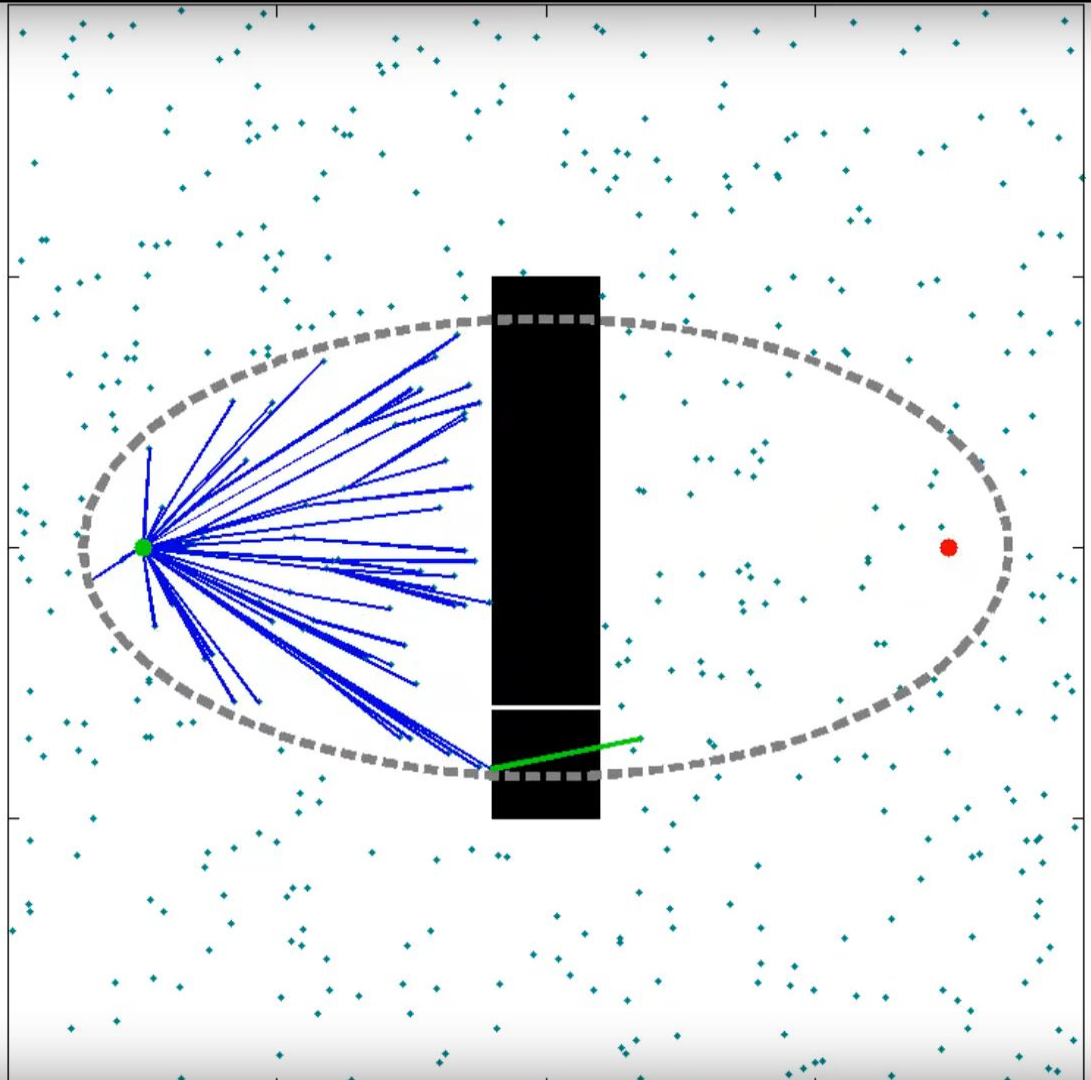
https://www.youtube.com/watch?v=TQIoCC48gp4



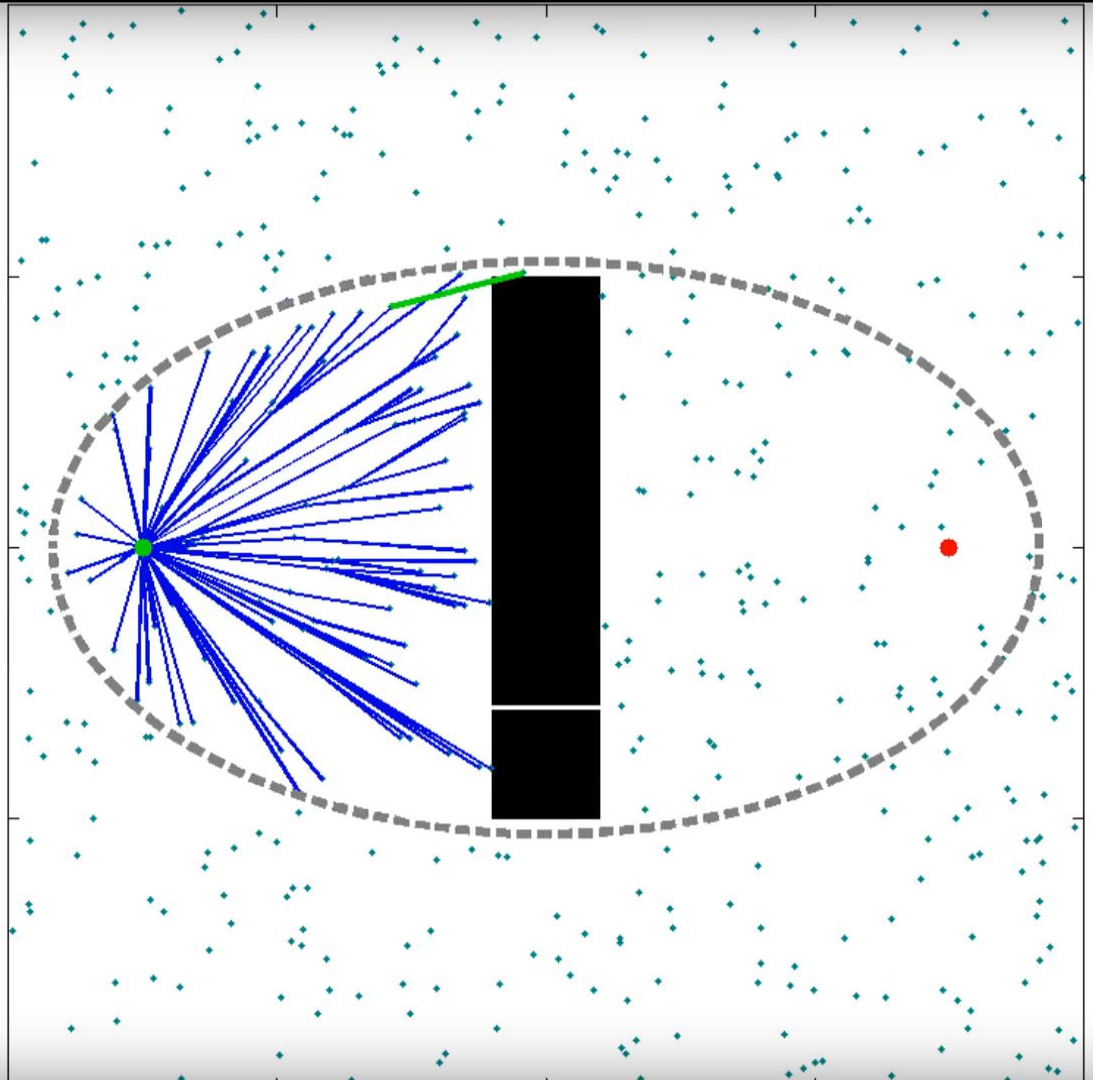
https://www.youtube.com/watch?v=TQIoCC48gp4



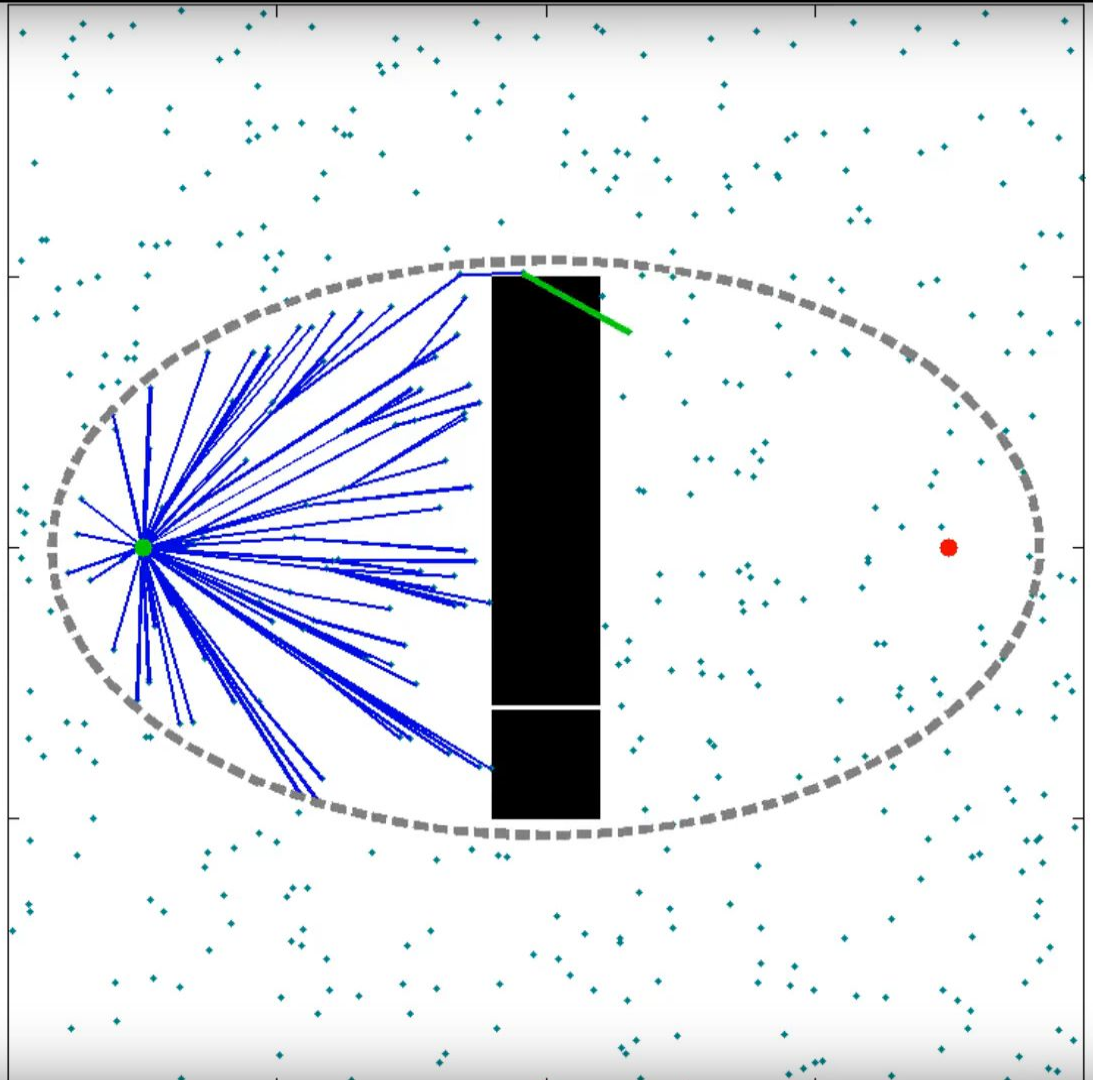
https://www.youtube.com/watch?v=TQIoCC48gp4



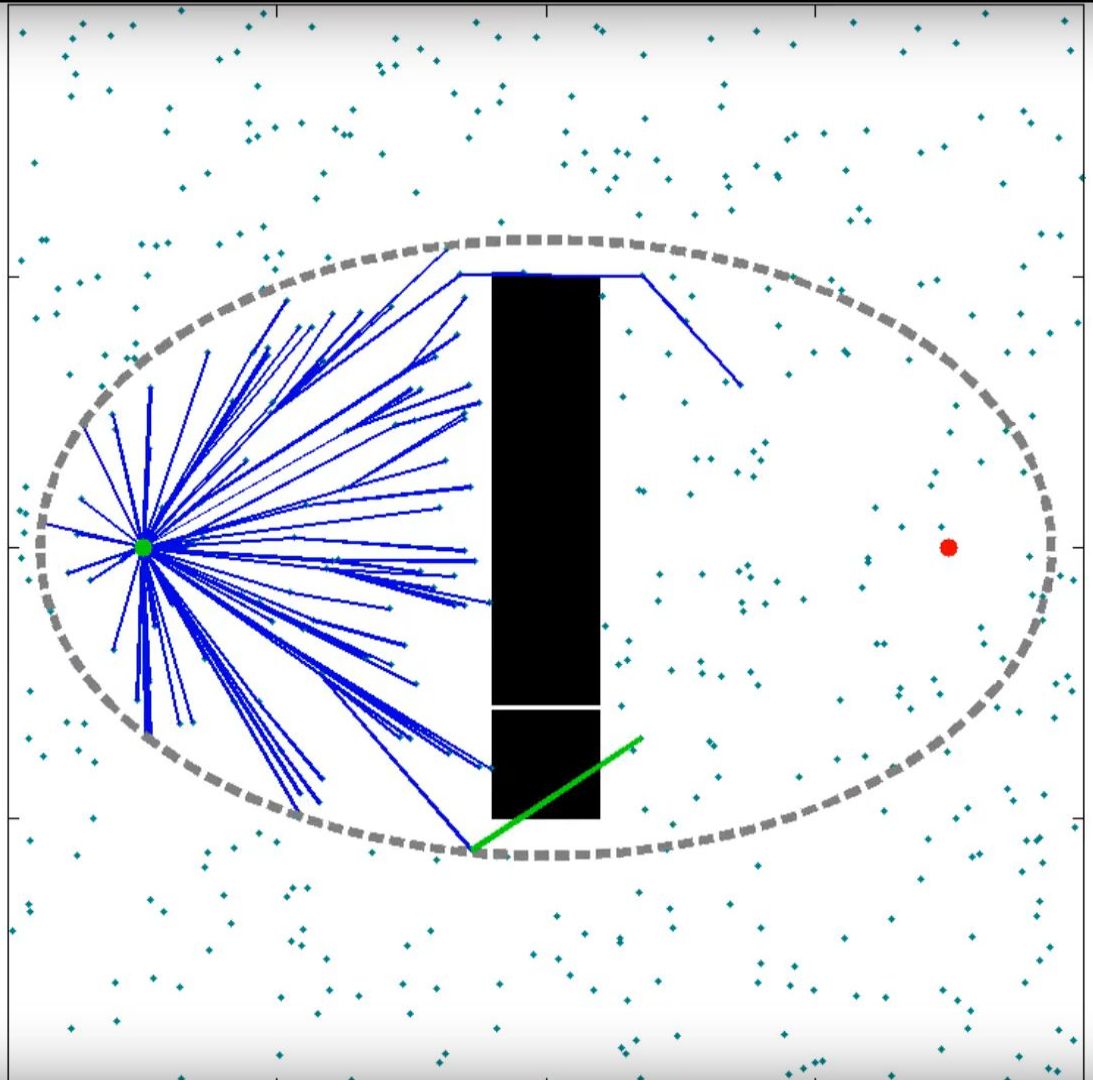
https://www.youtube.com/watch?v=TQIoCC48gp4



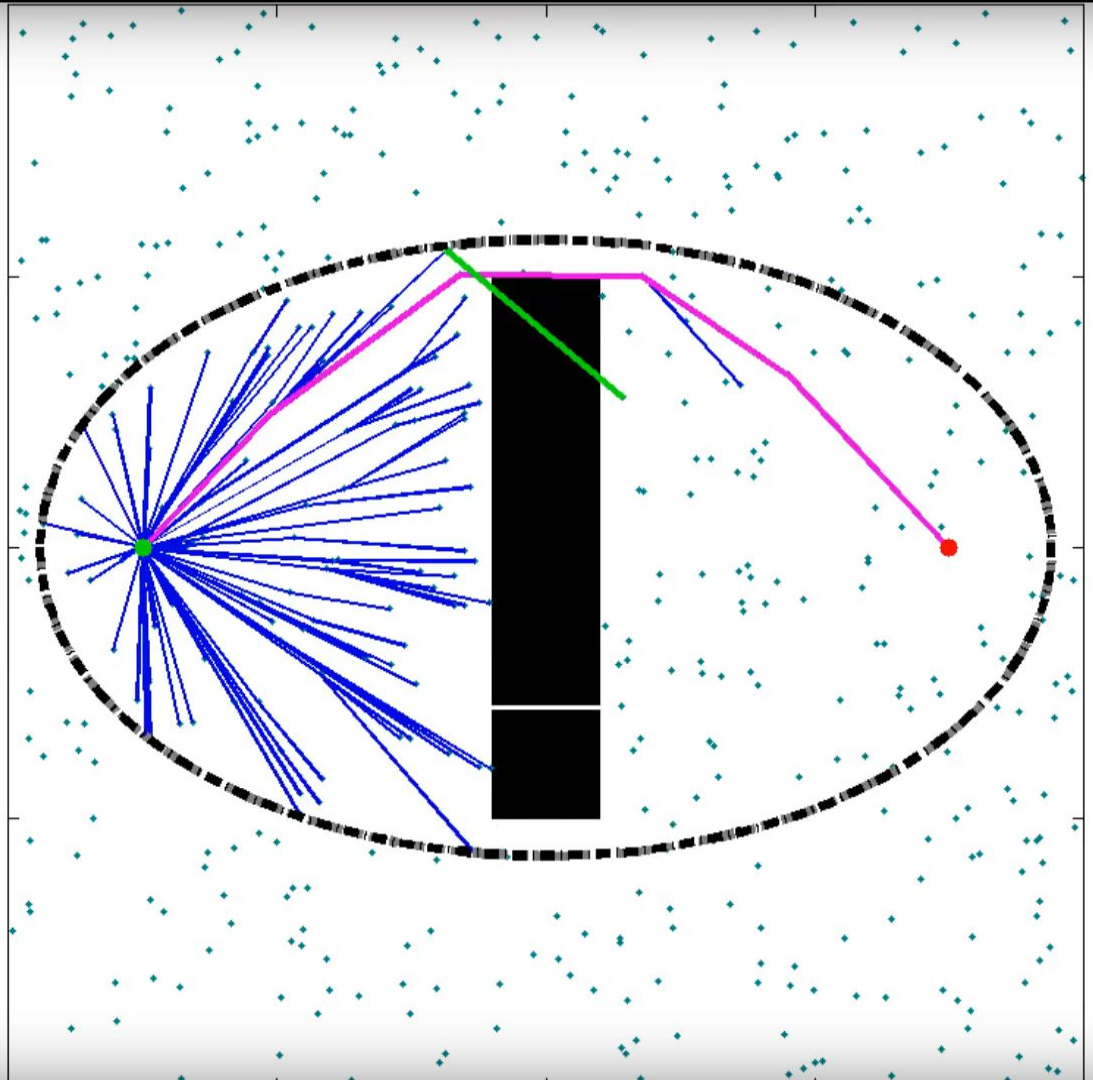
https://www.youtube.com/watch?v=TQIoCC48gp4



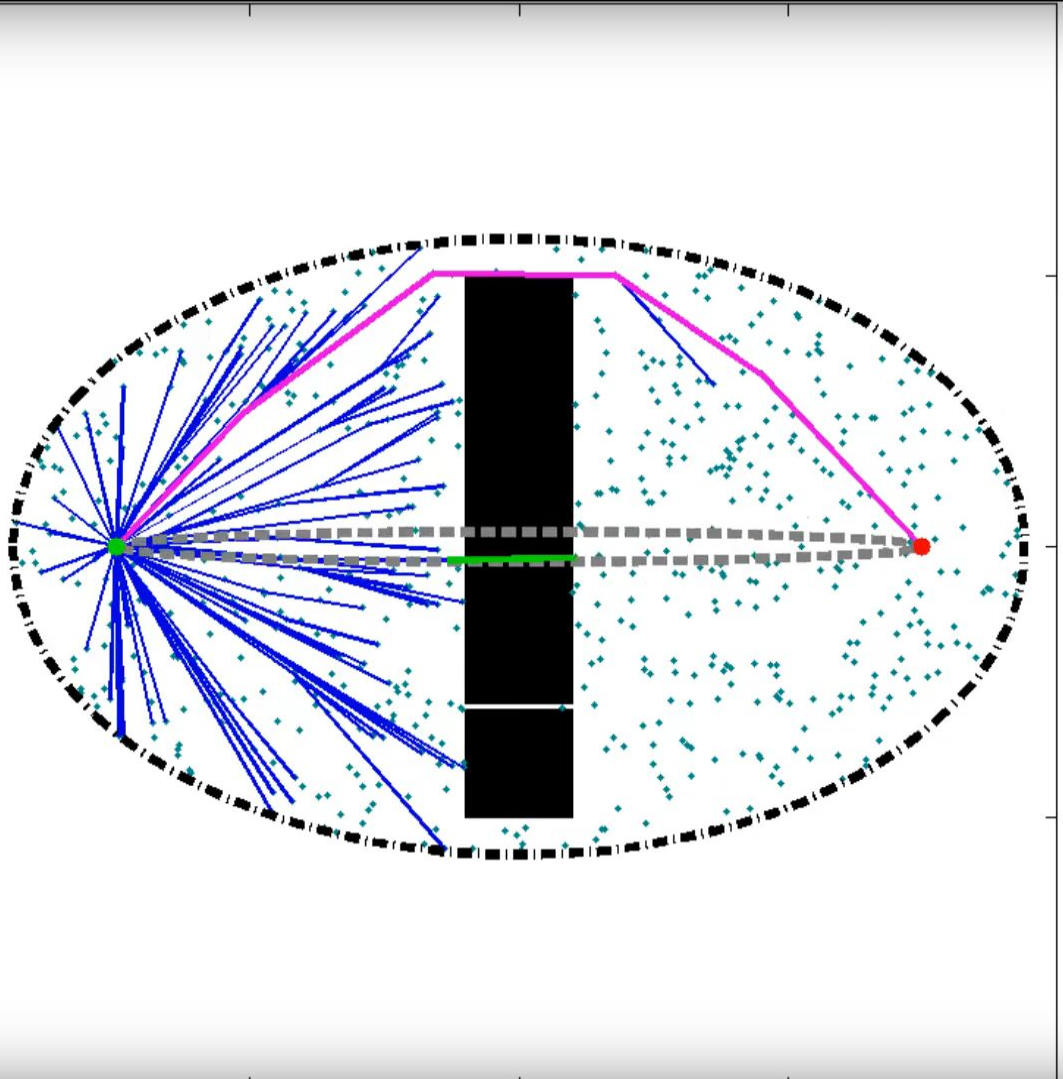
https://www.youtube.com/watch?v=TQIoCC48gp4



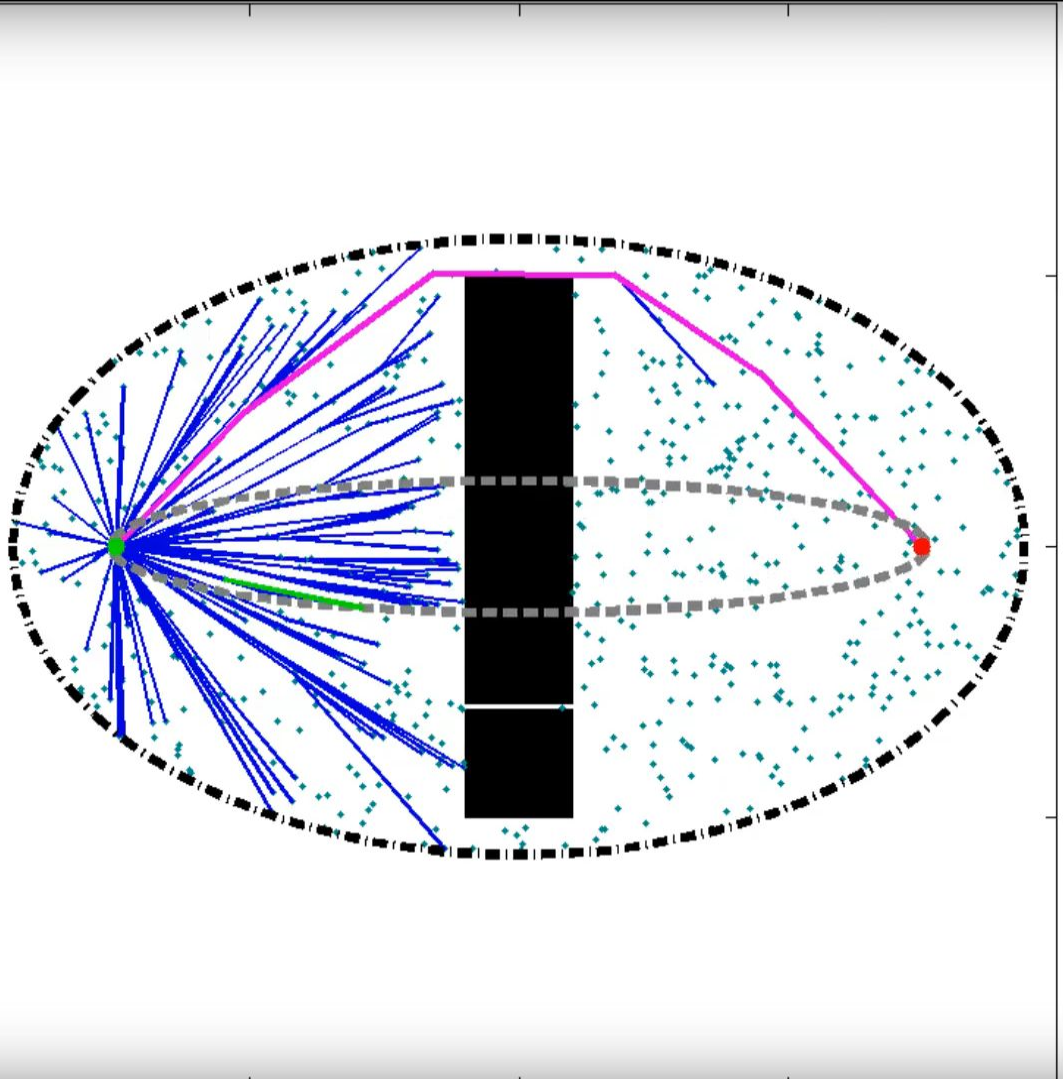
https://www.youtube.com/watch?v=TQIoCC48gp4



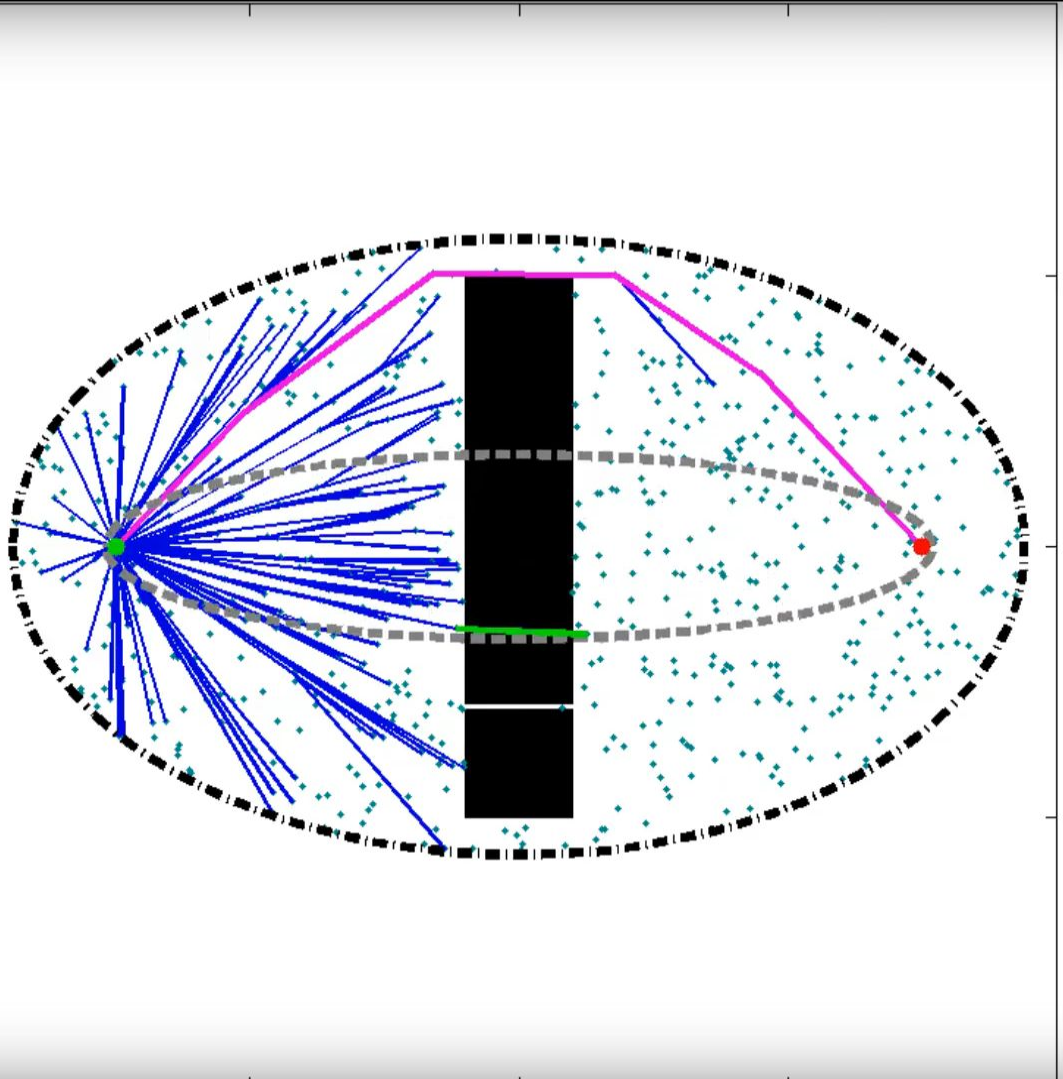
https://www.youtube.com/watch?v=TQIoCC48gp4



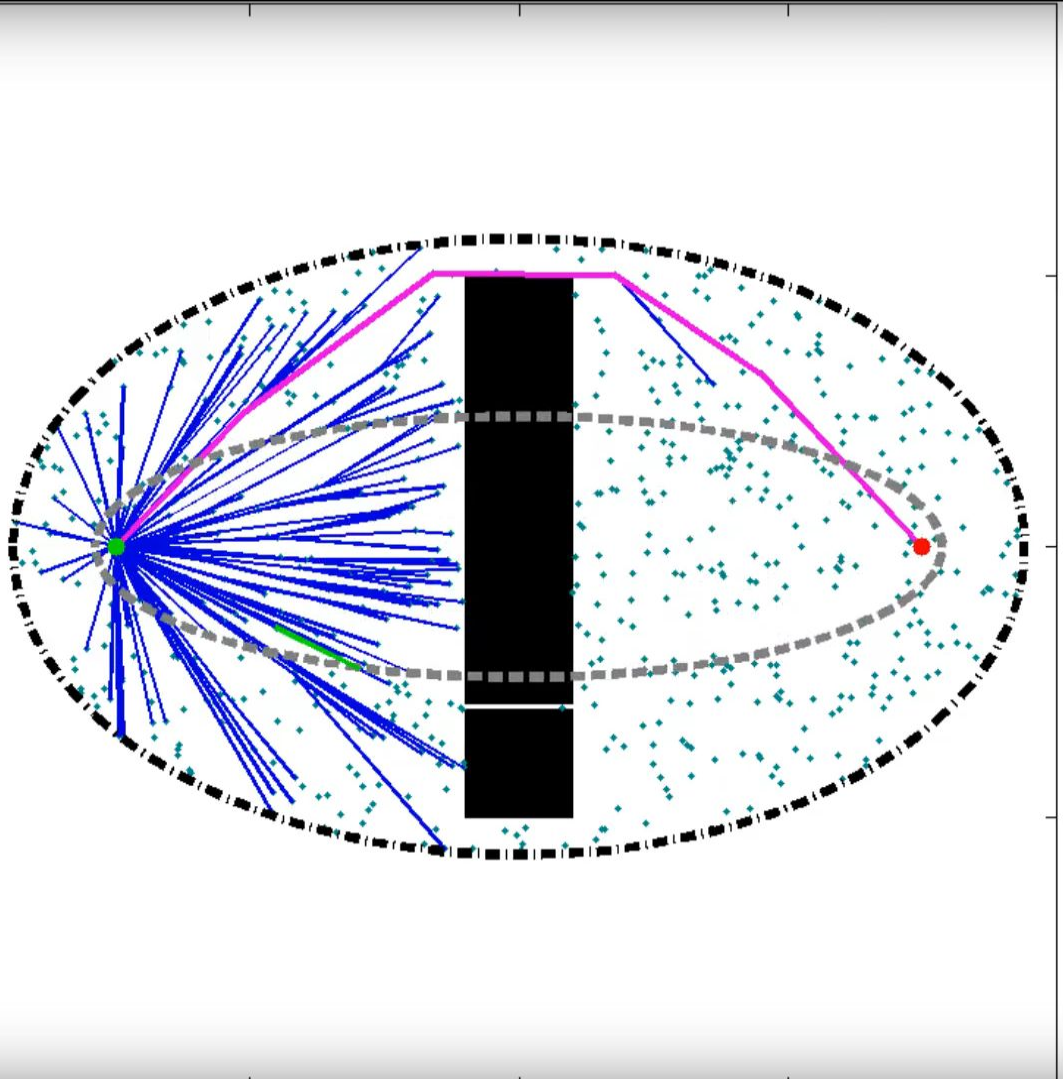
https://www.youtube.com/watch?v=TQIoCC48gp4



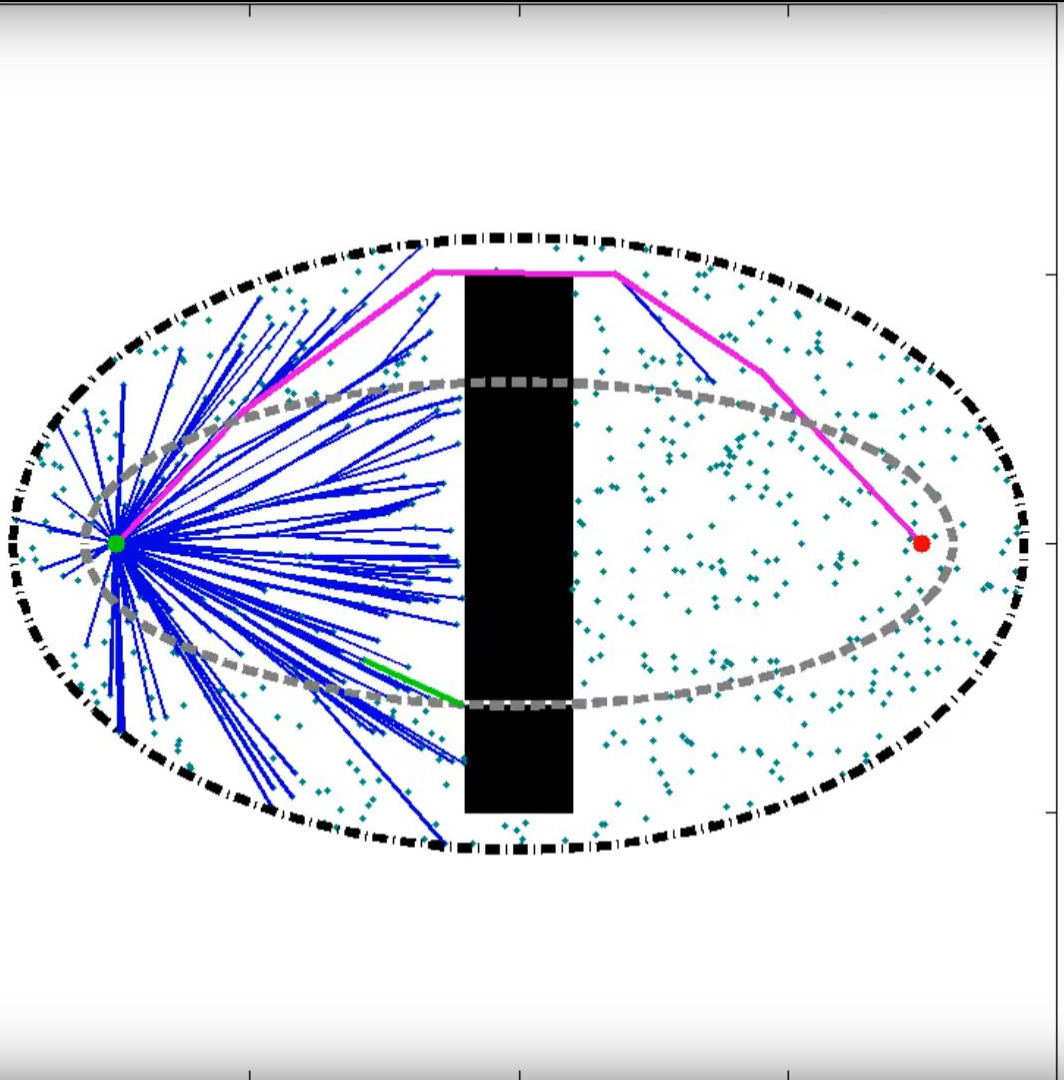
https://www.youtube.com/watch?v=TQIoCC48gp4



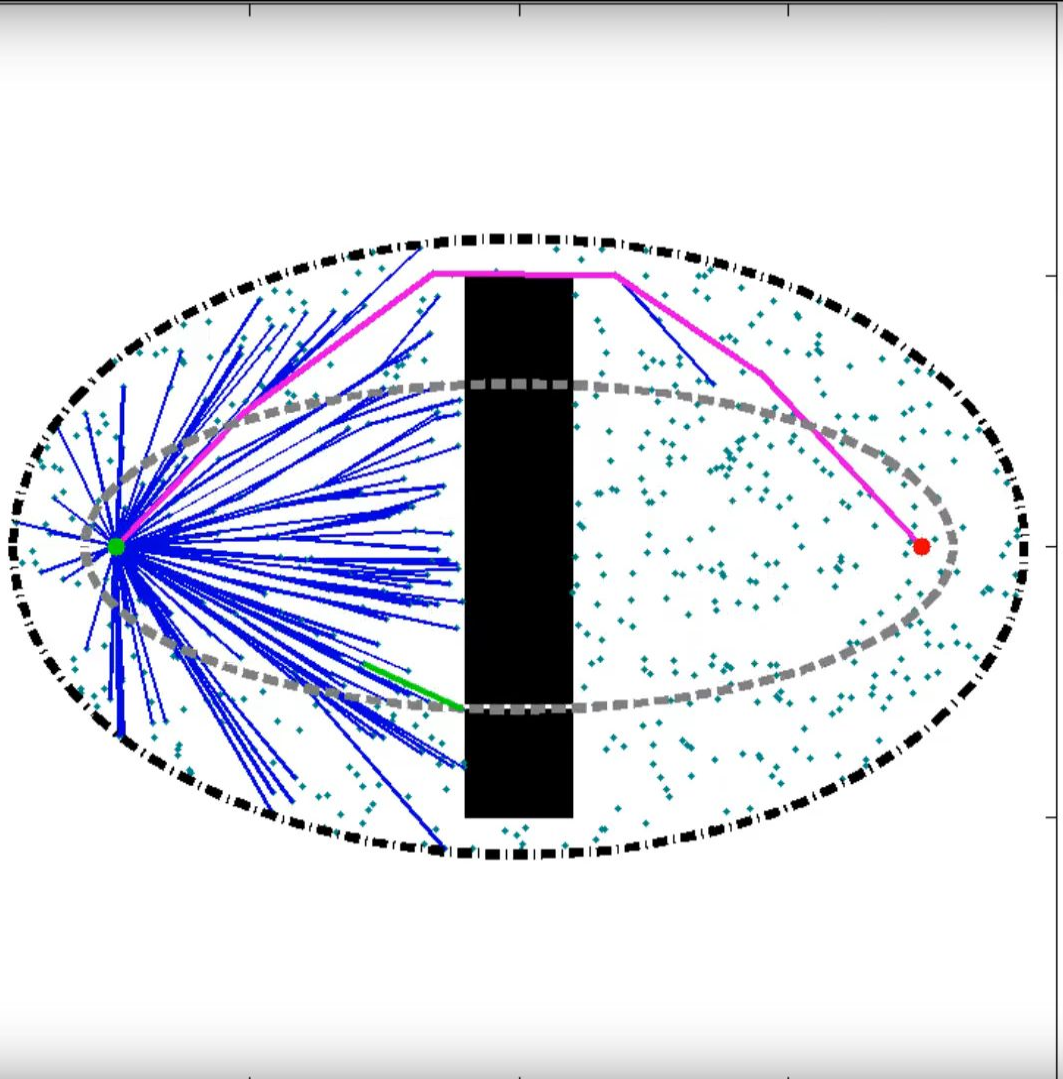
https://www.youtube.com/watch?v=TQIoCC48gp4



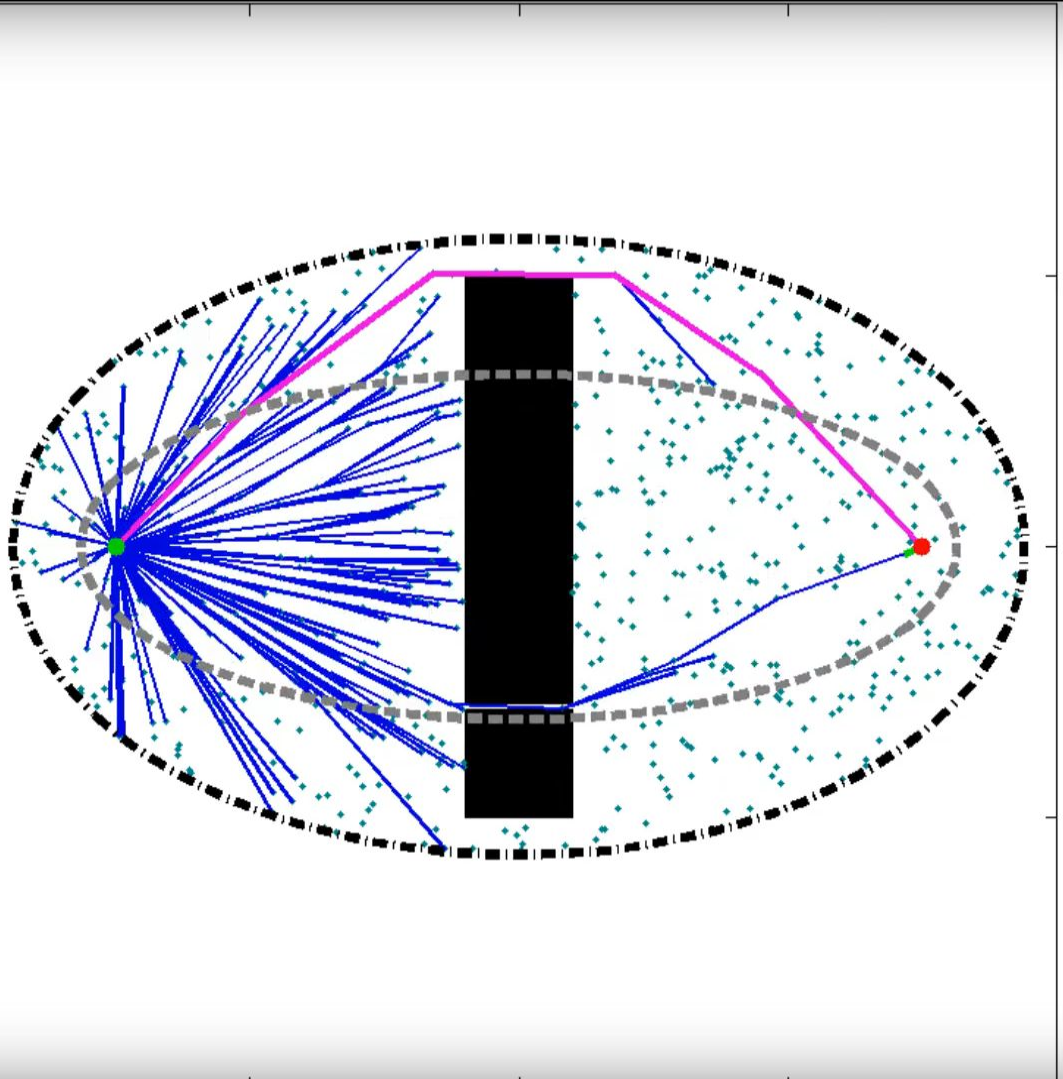
https://www.youtube.com/watch?v=TQIoCC48gp4



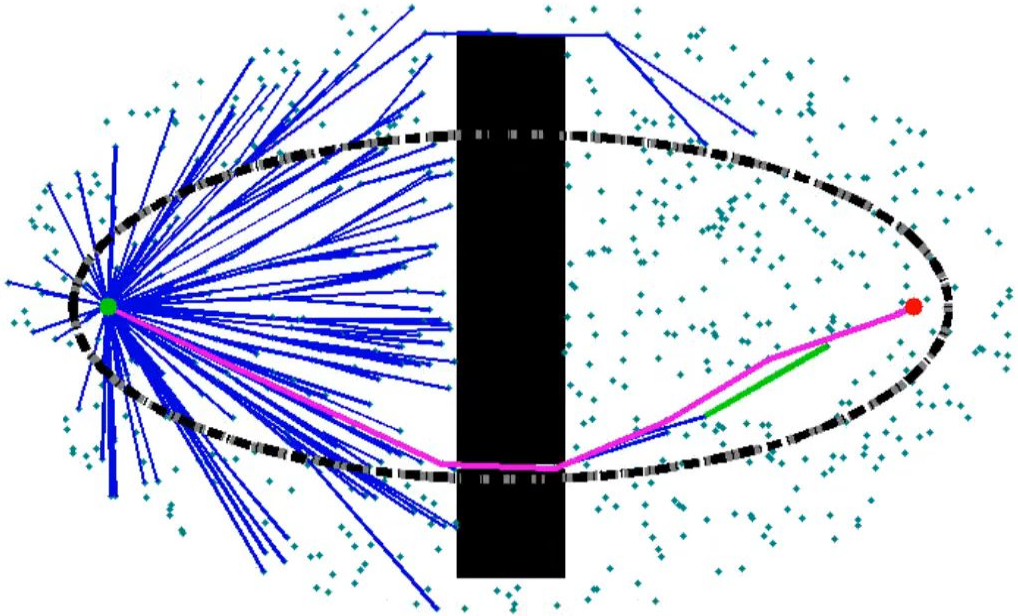
https://www.youtube.com/watch?v=TQIoCC48gp4



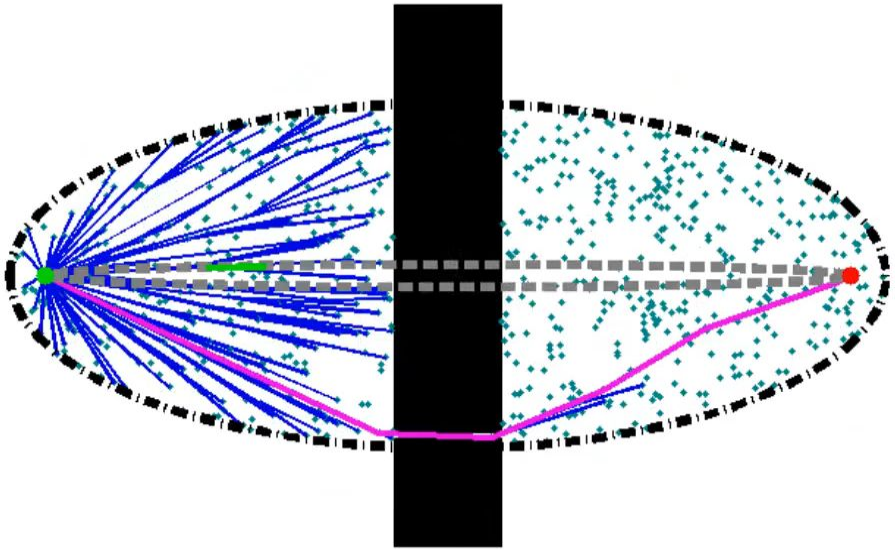
https://www.youtube.com/watch?v=TQIoCC48gp4

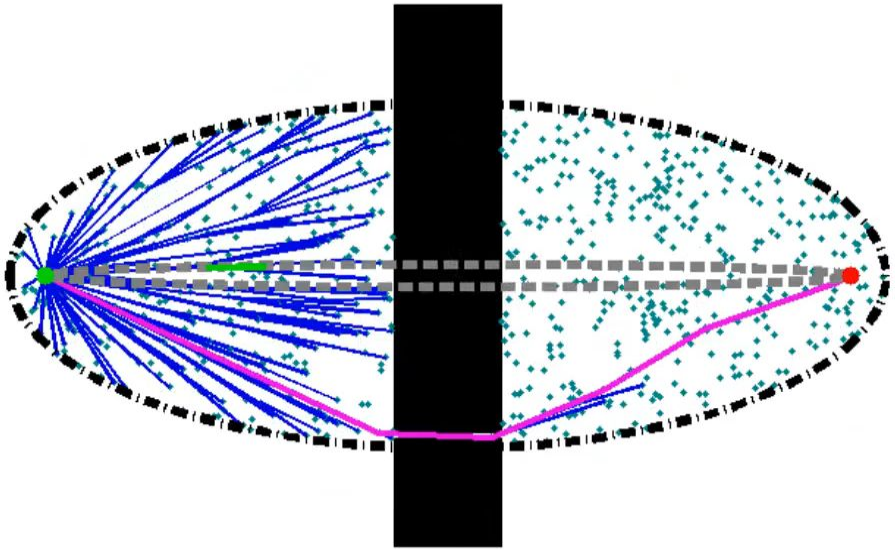


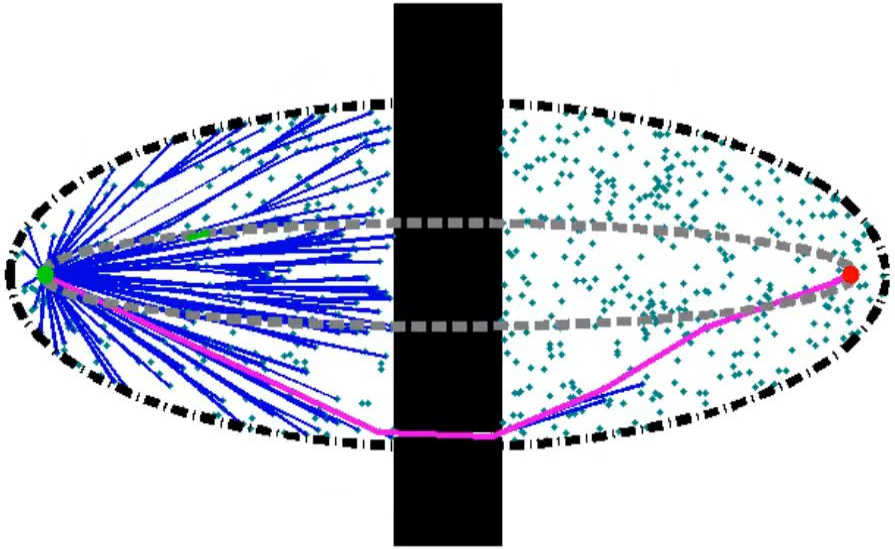
https://www.youtube.com/watch?v=TQIoCC48gp4



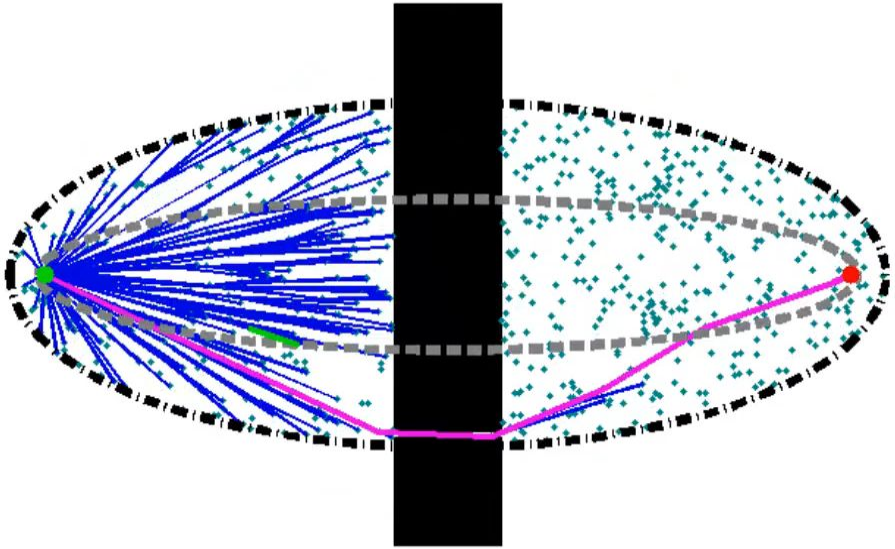
https://www.youtube.com/watch?v=TQIoCC48gp4



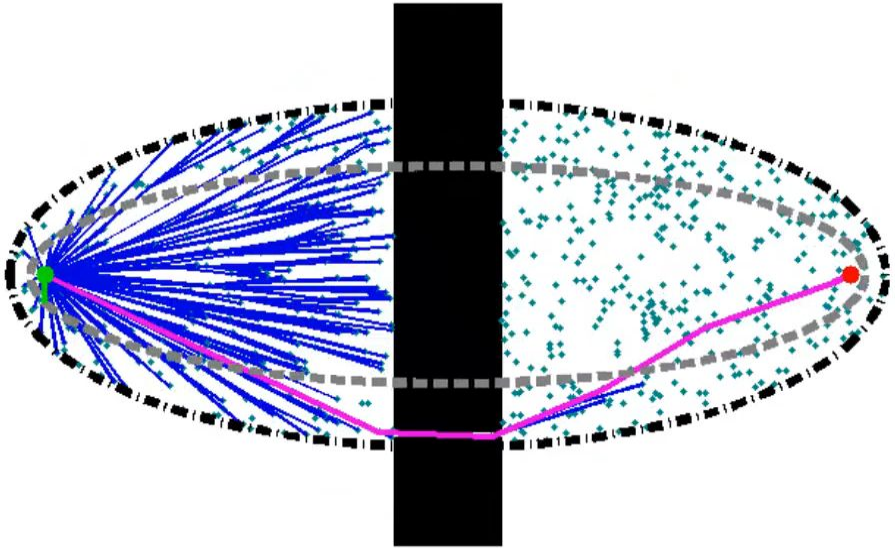




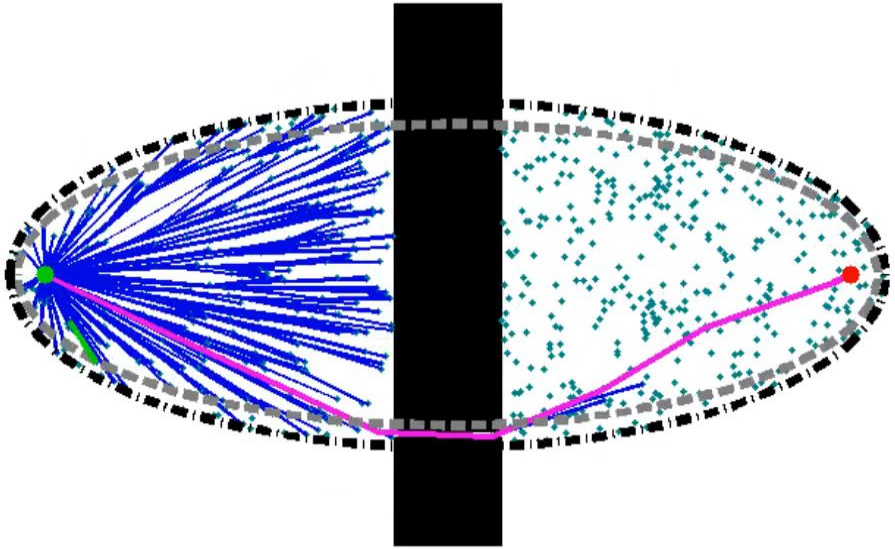
https://www.youtube.com/watch?v=TQIoCC48gp4



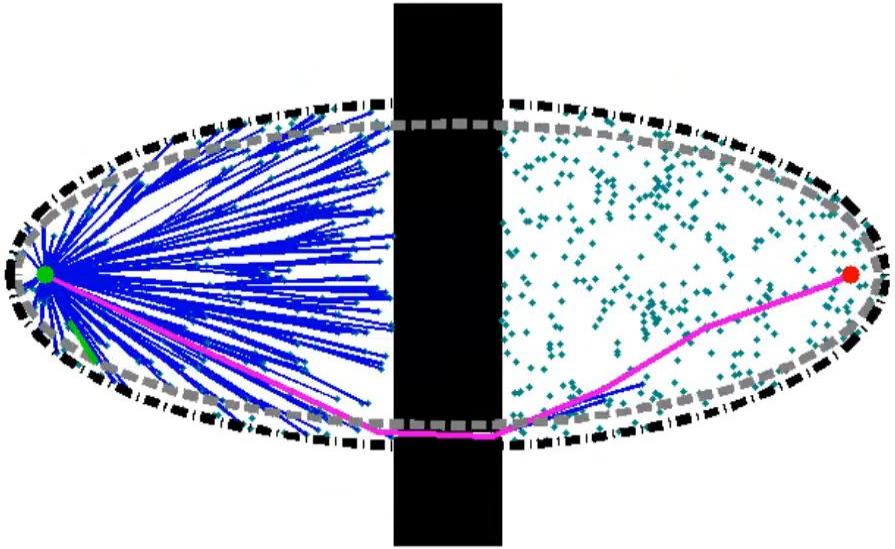
https://www.youtube.com/watch?v=TQIoCC48gp4



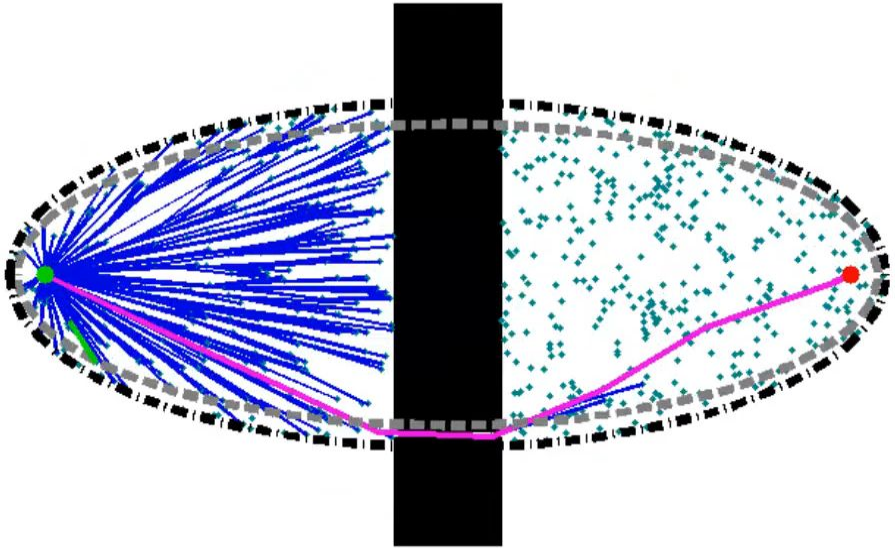
https://www.youtube.com/watch?v=TQIoCC48gp4



https://www.youtube.com/watch?v=TQIoCC48gp4



https://www.youtube.com/watch?v=TQIoCC48gp4



https://www.youtube.com/watch?v=TQIoCC48gp4

BIT* Details

- True edge cost calculation delayed
- Edge costs estimated and checked if it can improve the current solution
- Then checks collisions and differential constraints

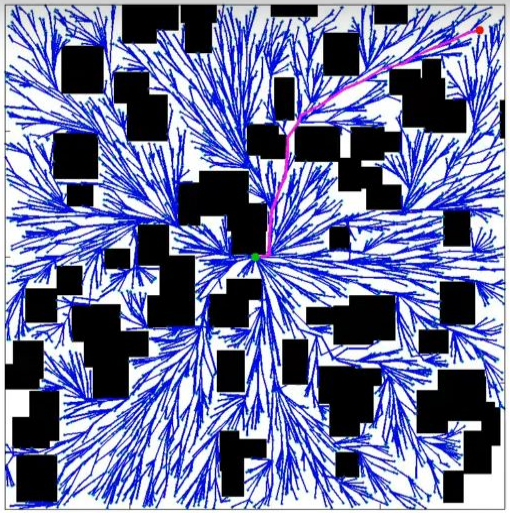
[Gammell15]

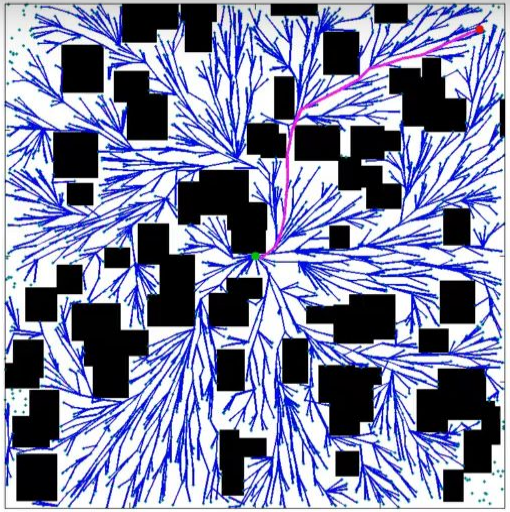
RRT*

$t = 00.427070s$

$c = 01.436973$

✓





FMT*

$t = 00.134621s$

$c = 01.450590$

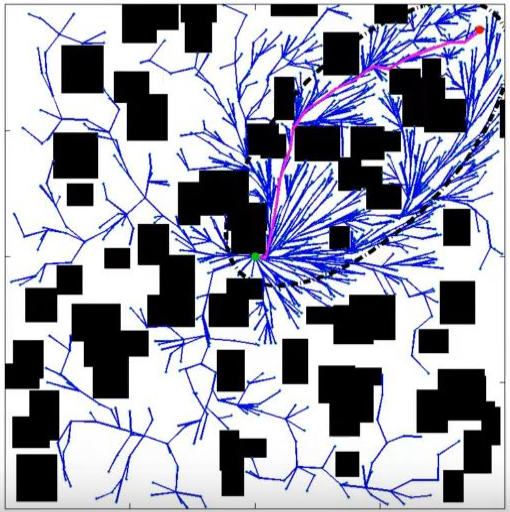
✓

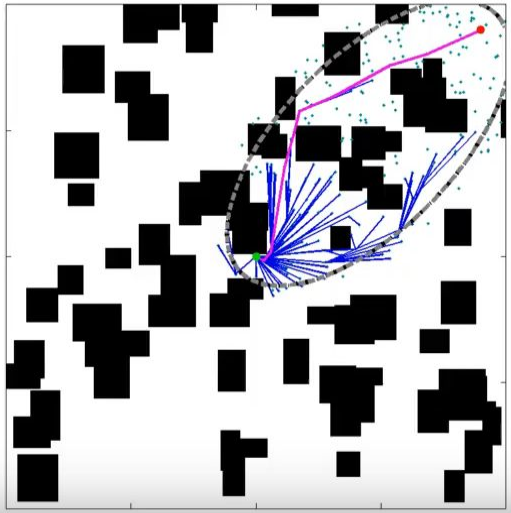
Informed RRT*

$t = 00.146614s$

$c = 01.447078$

✓





BIT*

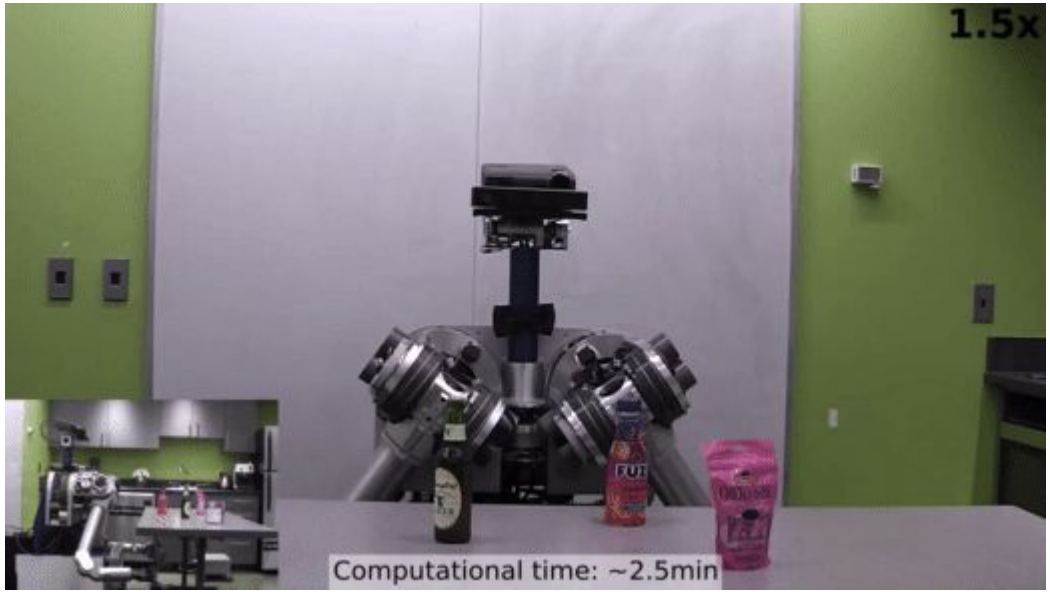
$t = 00.038243s$

$c = 01.447626$

✓

14 DOF Manipulation

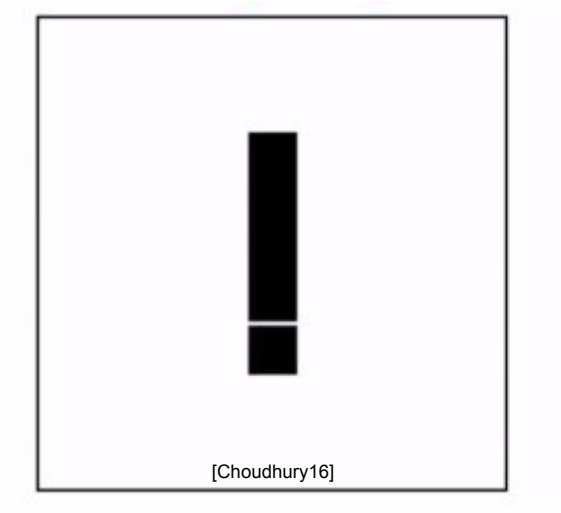
- Almost twice as likely to find a solution in allotted time
- Solution costs equal to or better than those of the other planners



[Gammell15]
https://www.youtube.com/watch?v=TQIoCC48gp4

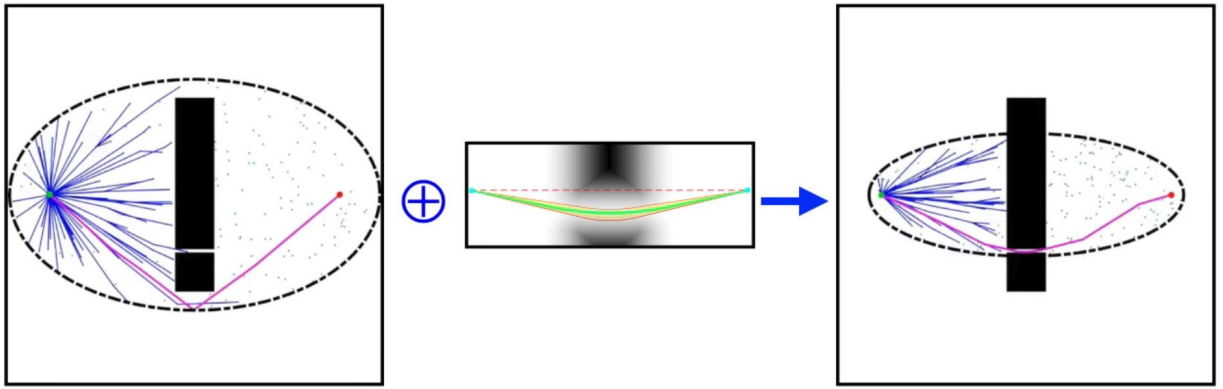
What’s next?

- Combining strenths from different methods drove the innovations of BIT*
- How could BIT* be improved?
  - Narrow passages are difficult to sample



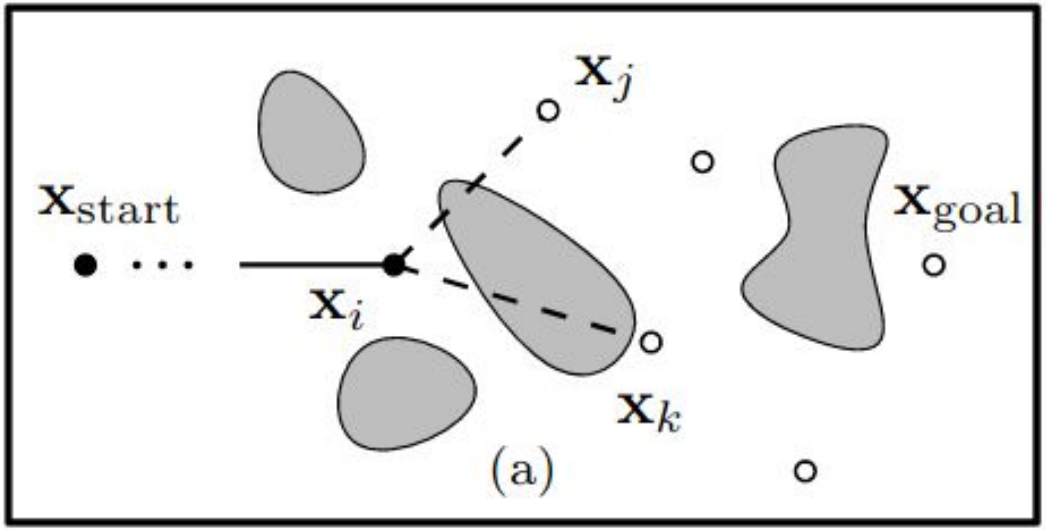
RABIT*

- *Regionally Accelerated Batch Informed Trees (RABIT*): A Framework to Integrate Local Information into Optimal Path Planning* (Choudhury et al. 2016)



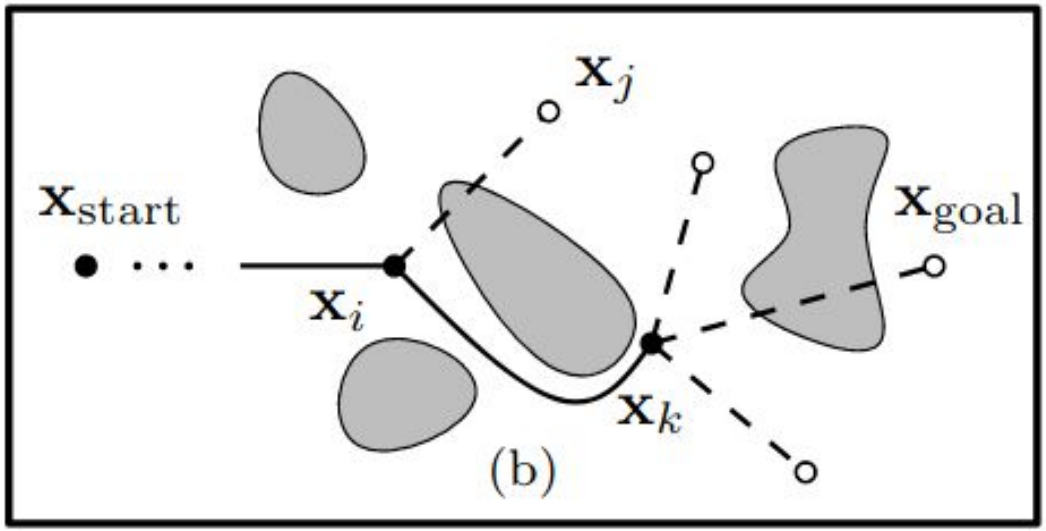
RABIT*

- BIT* method up until actual edge cost would be calculated



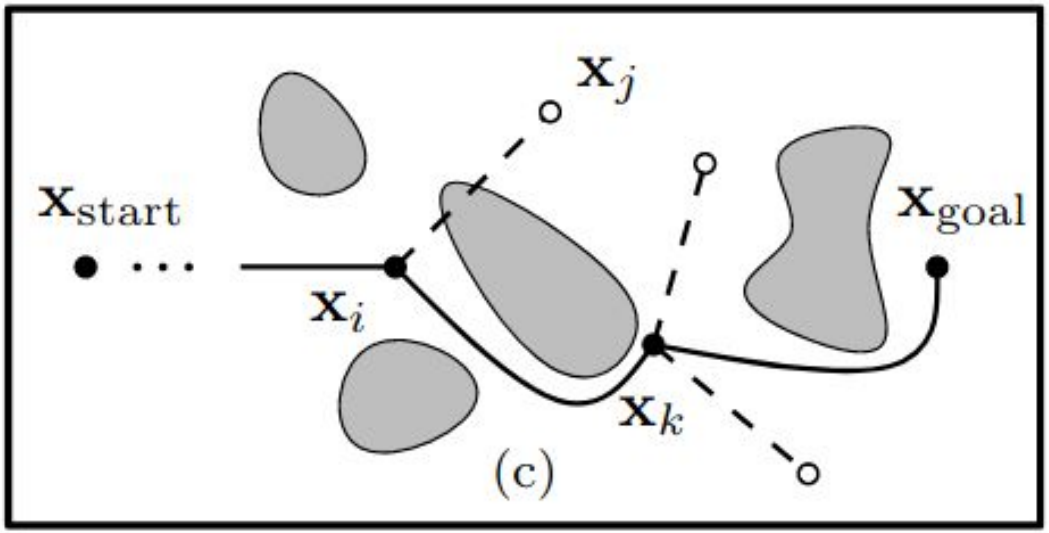
RABIT*

- Edge optimized using CHOMP and cost calculated



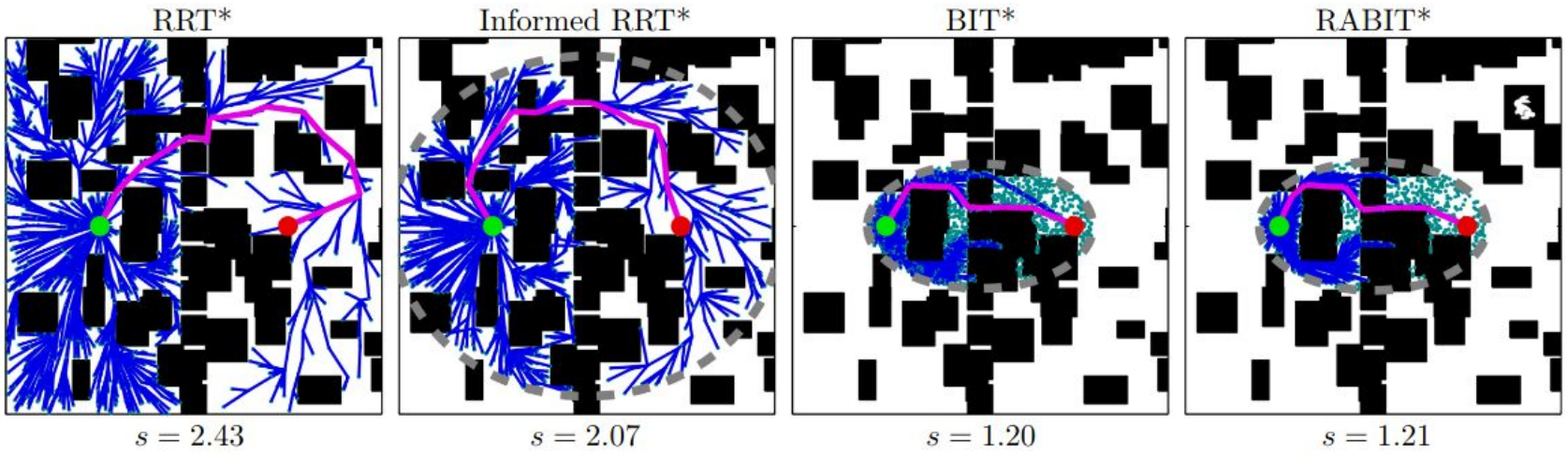
RABIT*

- Edge added if it improves the solution

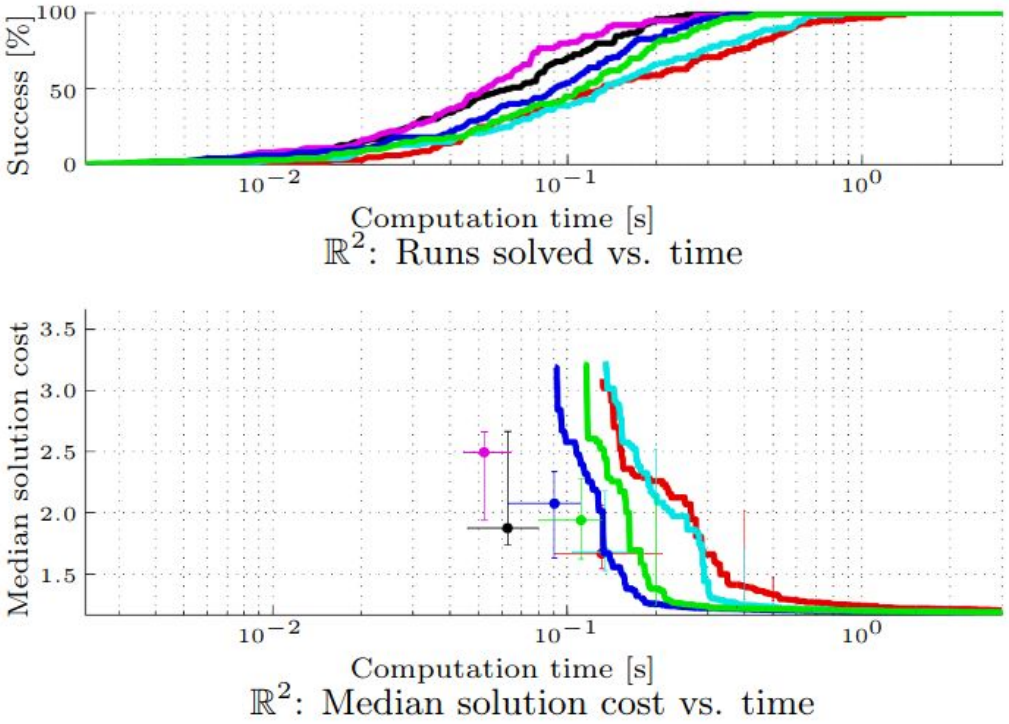


RABIT*: Comparison

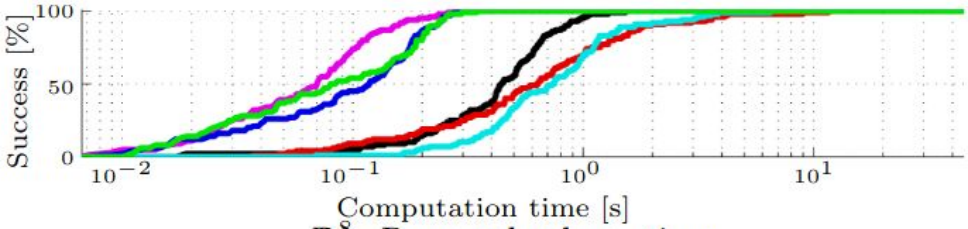
- 1.2 seconds computation time
- RABIT* comparable to BIT* in lower dimensions
- Performs better than BIT* when in higher-dimensional



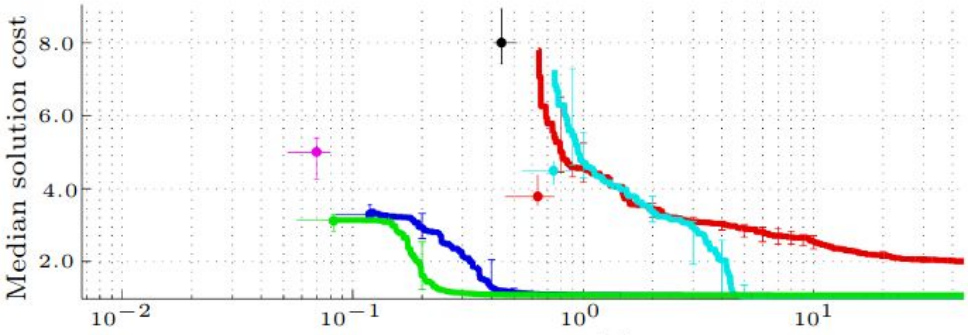
Two-dimensional Space

- 

Eight-dimensional Space

- 

$\mathbb{R}^8$: Runs solved vs. time



$\mathbb{R}^8$: Median solution cost vs. time

Latest BIT* News

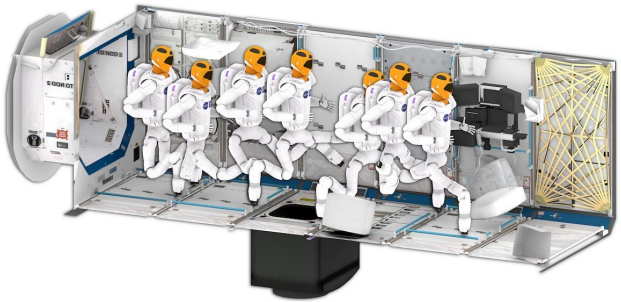
- ABIT* (Strub and Gammell 2020)
  - Uses advanced graph-search techniques to balance exploration and exploitation of the RGG

Outline

- Review
- FMT*
- Informed RRT*
- BIT*
- RABIT*
- **OMPL**
- Exercises

The Open Motion Planning Library

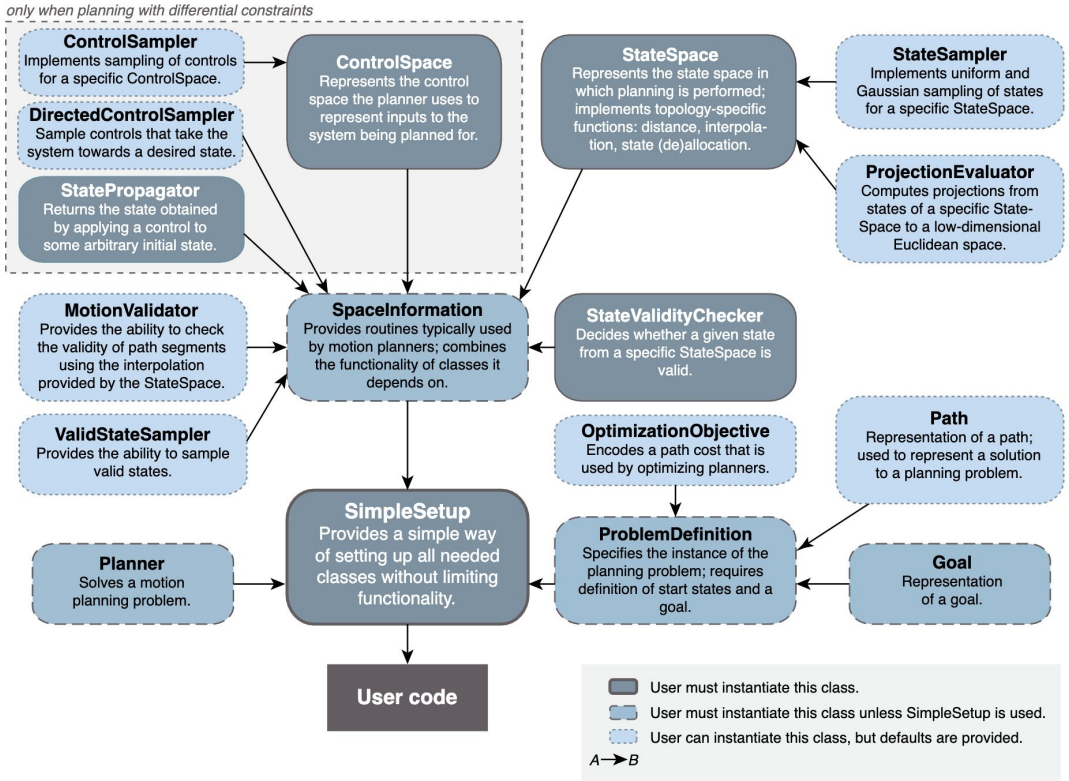
- Contains *abstracted* implementations of many state-of-the-art sampling-based algorithms.
- Widely used by industry and academia. Provides integration for MoveIt, MORSE, VREP, and ROS.
- To maintain abstraction, relies on users to provide domain-specific parts of the pipeline.
- In C++, but has python bindings.
- Good for benchmarking planners.
- Maintained by The Department of Computer Science at Rice University.



OMPL: https://ompl.kavrakilab.org/index.html

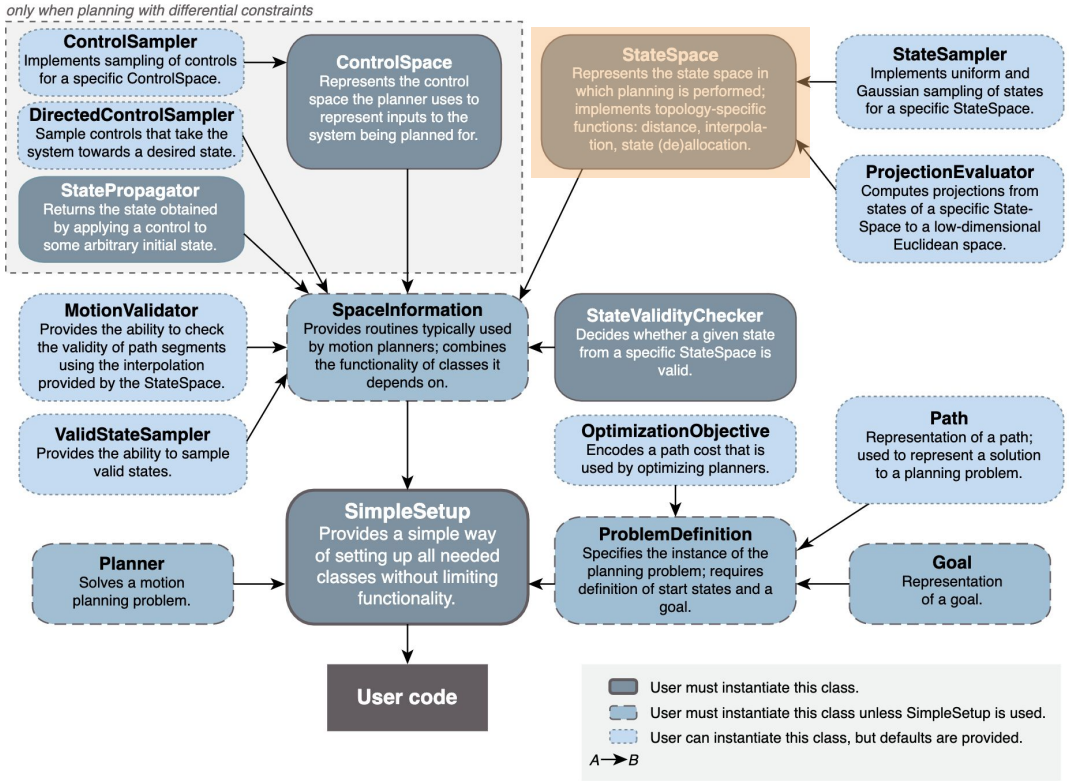
Disclaimer: This is not a complete review of the OMPL code-base but just the components the lab regularly uses!

OMPL::Overview



Poliety borrowed from https://ompl.kavrakilab.org/OMPL_Primer.pdf

OMPL::Overview

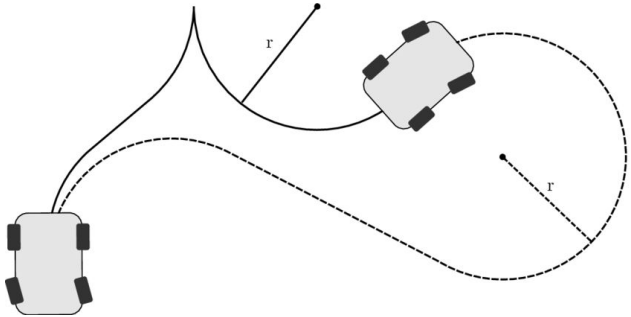


Proudly borrowed from https://ompl.kavrakilab.org/OMPL_Primer.pdf

OMPL::State-Space::General Case

OMPL provides some predefined state spaces:

- R^n : RealVectorSpace
- SO(2), SO(3) : Special Orthogonal groups
- SE(2), SE(3) : Special Euclidean groups
- Time
- Dubins, ReedsShepp



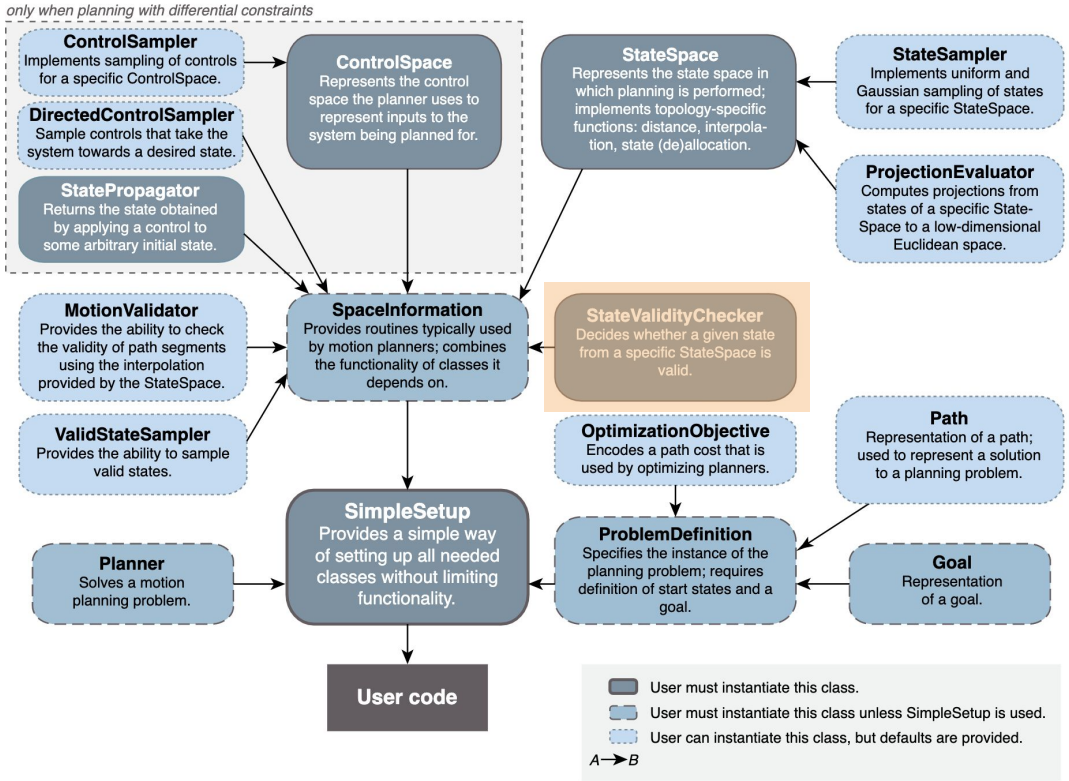
Dubins (dashed) and Reeds-Shepp (solid) curves [Kurzer, Karl.]

More info: https://ompl.kavrakilab.org/workingWithStates.html

OMPL::State-Space::Our Case

- We will assume UAVs are operating in (**x,y,z,ψ**) state space (design choice).
- Good assumption given differential flatness of UAVs. [Cowling et al. `08]
- OMPL allows construction of custom state-space by combining existing state-spaces

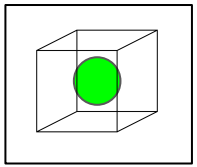
OMPL::Overview

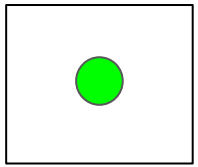


Stealthily borrowed from https://ompl.kavrakilab.org/OMPL_Primer.pdf

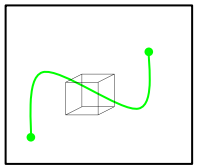
OMPL::Validity-Checking::General Case

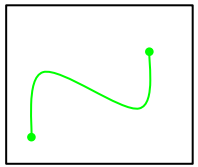
- Makes OMPL independent of the map and vehicle representation.
- State Validity Checking: Boolean to determine if a particular state is valid.


: Invalid State


: Valid State

- Motion Validity Checking: Boolean to determine if motion between states is valid.


: Invalid Motion


: Valid Motion

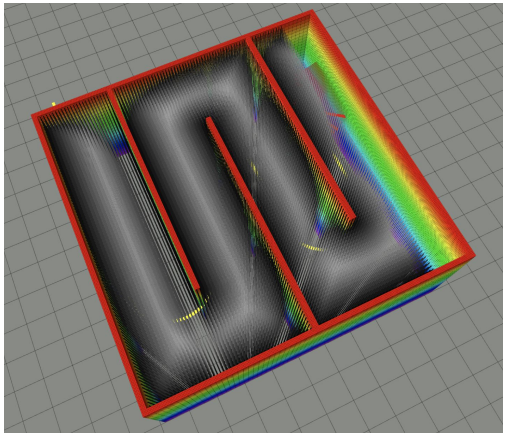
More info: https://ompl.kavrakilab.org/stateValidation.html

OMPL::Validity-Checking::Our Case

- Integration with core_stacks map representations can be achieved. Currently support for octomaps is available in core_stacks. Support for other map representations can be added.

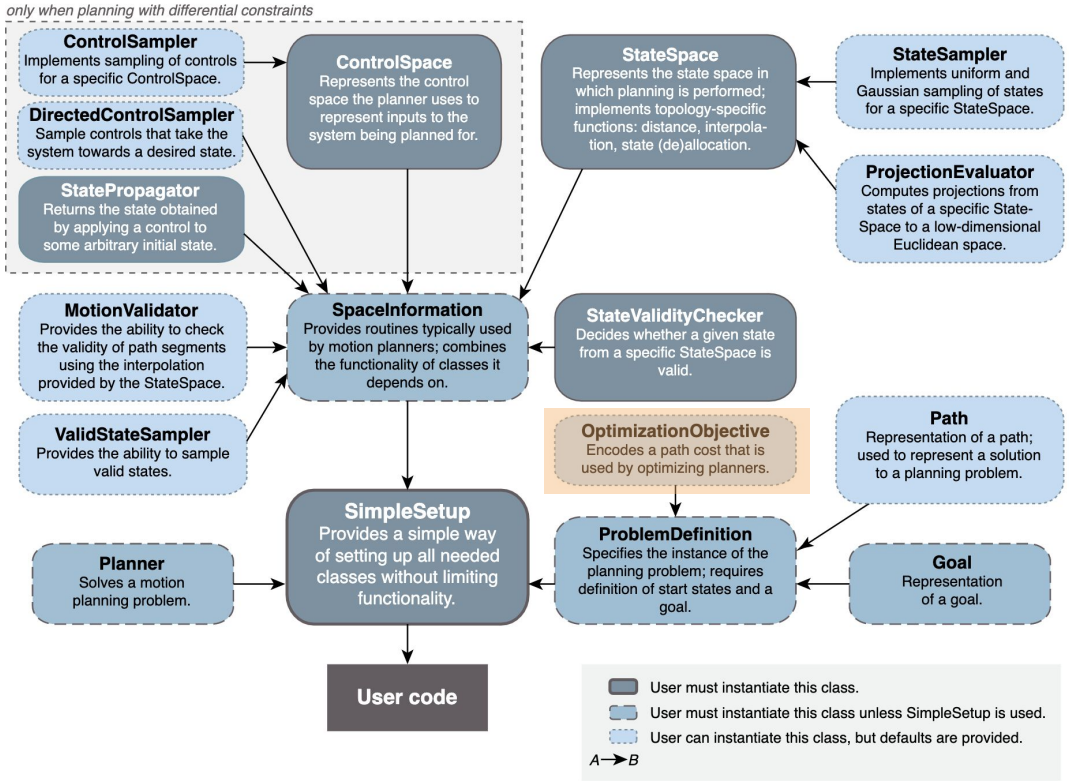
- State Validity checking is mandatory for obstacle avoidance.

- Default Motion Validity checking that uses a discrete motion checker can be used by specifying a state validity checking resolution.



https://bitbucket.org/castacks/core_octomap_map_representation/src/master/

OMPL::Overview



Dutifully borrowed from https://ompl.kavrakilab.org/OMPL_Primer.pdf

OMPL::Objectives::General Case

- OMPL provides some predefined objectives.
  - Path length
  - Minimum path clearance
  - General state cost integral
  - Mechanical work

- A custom objective function that returns a double can be specified.
- State-to-State heuristics can also be supplied here.
- Limited support for multiobjective functions. Can support weighted sum. No MinMax support.

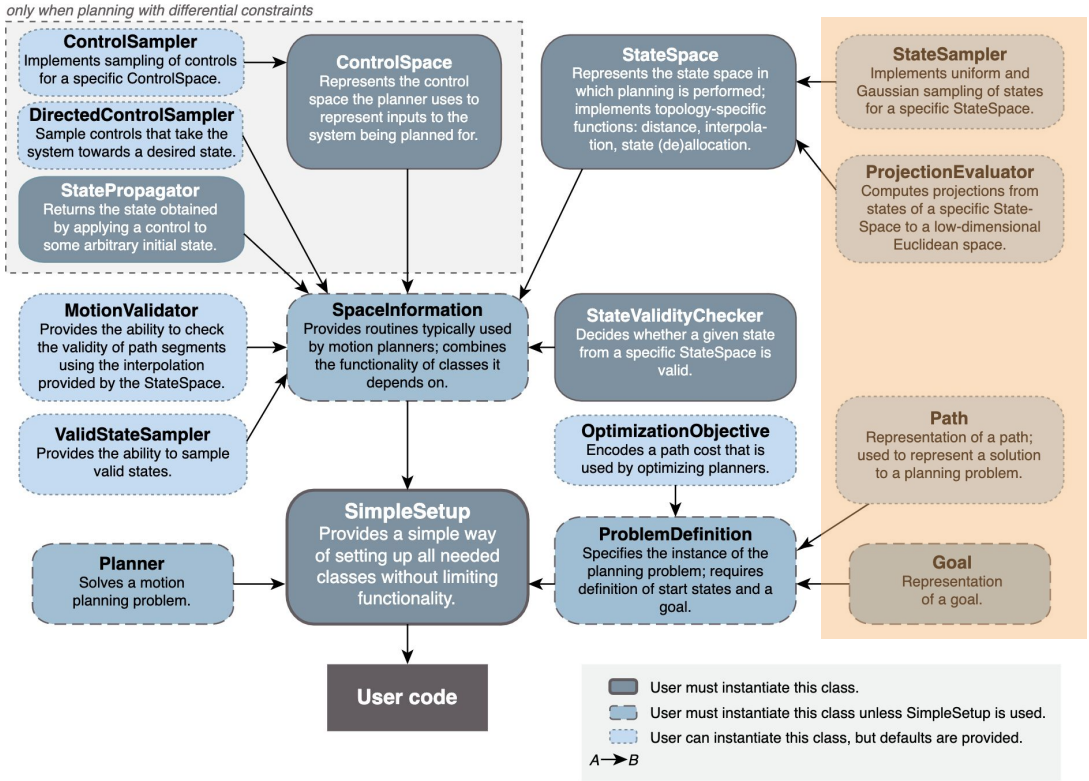
More info: https://ompl.kavrakilab.org/optimizationObjectivesTutorial.html

OMPL::Objectives::Our Case

- Objectives we are usually interested in:
  - **Path Length** (Our focus for the tutorial)
  - Time
  - Smoothness
  - Path Clearance
  - Energy

More info: https://ompl.kavrakilab.org/optimizationObjectivesTutorial.html

OMPL::Overview

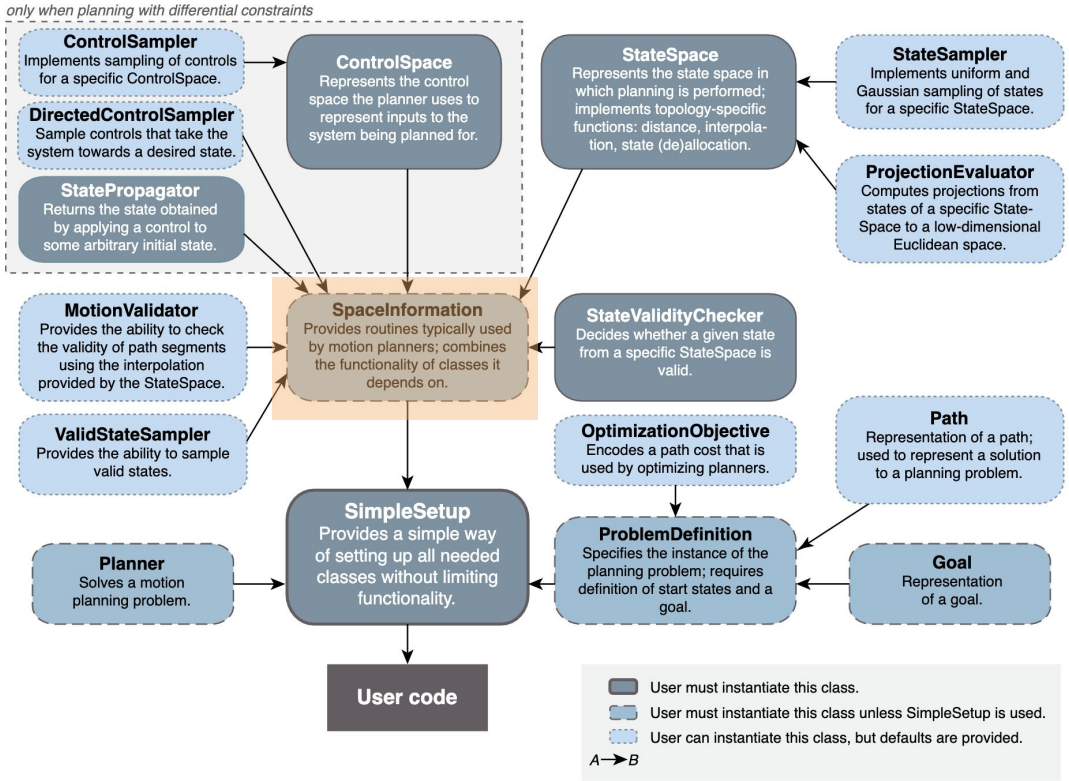


Unofficially borrowed from https://ompl.kavrakilab.org/OMPL_Primer.pdf

OMPL::Things we did not cover

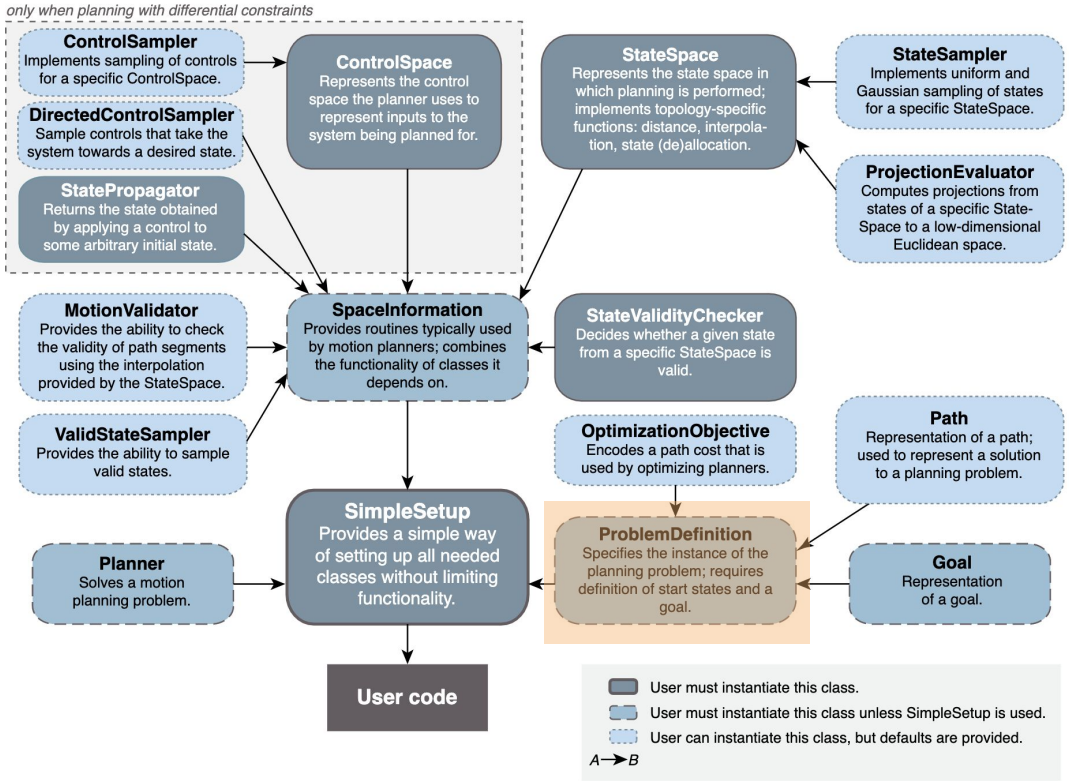
- Goals:
  - OMPL supports point-goals and goal regions. A custom class can also be defined.
- Paths:
  - OMPL has an implementation for representing paths. Basic methods like append(), interpolate(), cost(), etc are provided. No visualization support.
  - We have our own implementation for paths with domain specific methods and visualization tools.
- State (Informed?) Samplers
- Termination conditions (Time, # Iter, etc ...)
- Planning in control space (differential constraints)

OMPL::Overview



Quietly borrowed from https://ompl.kavrakilab.org/OMPL_Primer.pdf

OMPL::Overview

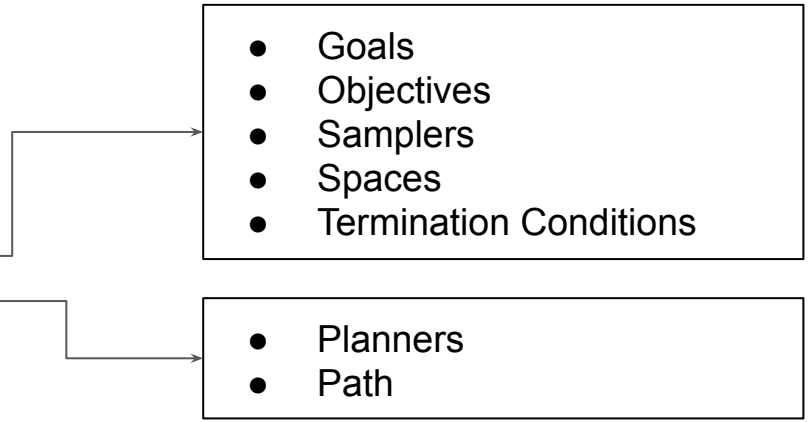


Shamelessly borrowed from https://ompl.kavrakilab.org/OMPL_Primer.pdf

OMPL::Let’s build a planning instance

Step 1. Get your namespaces!

```
namespace ob = ompl::base;
namespace og = ompl::geometric;
```

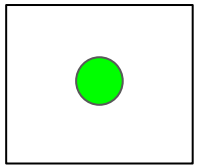


Based on : https://ompl.kavrakilab.org/geometricPlanningSE3.html

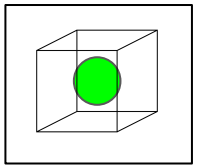
OMPL::Let’s build a planning instance

Step 2. Select your map representation and state validity checker

```
bool isStateValid(const ob::State *state)
```



: Valid State



: Invalid State

Based on : https://ompl.kavrakilab.org/geometricPlanningSE3.html

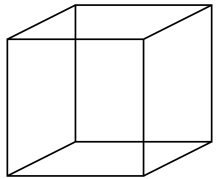
OMPL::Let's build a planning instance

Step 3. Select your state space and setup bounds

```
void plan()
{
    // construct the state space we are planning in
    auto space(std::make_shared<ob::SE3StateSpace>());
```

```
  ob::RealVectorBounds bounds(3);
  bounds.setLow(-1);
  bounds.setHigh(1);

  space->setBounds(bounds);
```



Based on : https://ompl.kavrakilab.org/geometricPlanningSE3.html

OMPL::Let’s build a planning instance

Step 4. Construct your Space Information class and provide other details

```
auto si(std::make_shared<ob::SpaceInformation>(space));
```

```
si->setStateValidityChecker(isStateValid);
```

.

Based on : https://ompl.kavrakilab.org/geometricPlanningSE3.html

OMPL::Let’s build a planning instance

Step 5. Get Start and Goal

```
ob::ScopedState<> start(space);
start.random();
```

```
ob::ScopedState<> goal(space);
goal.random();
```

Based on : https://ompl.kavrakilab.org/geometricPlanningSE3.html

OMPL::Let’s build a planning instance

Step 6. Construct your Problem Definition class and provide other details

```
auto pdef(std::make_shared<ob::ProblemDefinition>(si));
```

```
pdef->setStartAndGoalStates(start, goal);
```

Based on : https://ompl.kavrakilab.org/geometricPlanningSE3.html

OMPL::Let’s build a planning instance

Step 7. Select your planner and bring everything together

```
auto planner(std::make_shared<og::RRTConnect>(si));
```

→ Select Planner and provide the space information.

```
planner->setProblemDefinition(pdef);
```

→ Provide the problem definition

```
planner->setup();
```

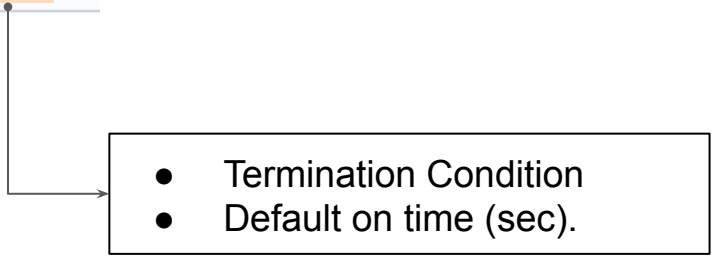
→ Prime the planner

Based on : https://ompl.kavrakilab.org/geometricPlanningSE3.html

OMPL::Let’s build a planning instance

Step 8. Solve!!

```
ob::PlannerStatus solved = planner->ob::Planner::solve(1.0);
```



Based on : https://ompl.kavrakilab.org/geometricPlanningSE3.html

OMPL::Let's build a planning instance

Step 9. Post-process the solution

```
if (solved)
{
    // get the goal representation from the problem definition (not the same as the goal state)
    // and inquire about the found path
    ob::PathPtr path = pdef->getSolutionPath();
    std::cout << "Found solution:" << std::endl;

    // print the path to screen
    path->print(std::cout);
}
```

Based on : https://ompl.kavrakilab.org/geometricPlanningSE3.html

References

[Gammell15]: Batch Informed Trees (BIT*): Sampling-based Optimal Planning viathe Heuristically Guided Search of Implicit Random Geometric Graphs, https://arxiv.org/pdf/1405.5848.pdf, https://www.youtube.com/watch?v=TQIoCC48gp4

[Gammell14]: Informed RRT*: Optimal Sampling-based Path Planning Focused viaDirect Sampling of an Admissible Ellipsoidal Heuristic, https://www.youtube.com/watch?v=nsl-5MZfwu4

Kurzer, Karl. "Path planning in unstructured environments: A real-time Hybrid A* implementation for fast and deterministic path generation for the KTH Research Concept Vehicle." (2016).

Cowling, Ian D., et al. "A prototype of an autonomous controller for a quadrotor UAV." *2007 European Control Conference (ECC)*. IEEE, 2007.

[Lopez]: Ph.D. Dissertation - Reactive Evolutionary Path Planning for Autonomous Surface Vehicles in Lake Environments, Mario Arzamendia Lopez

[Amit]: http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

[Choudhury16]: Regionally Accelerated Batch Informed Trees (RABIT*): A Framework to Integrate Local Information into Optimal Path Planning, https://www.youtube.com/watch?v=mqq-DW36jSo

[Janson15]: Fast Marching Tree: a Fast Marching Sampling-Based Method for Optimal Motion Planning in Many Dimensions*, Fast Marching Trees: a Fast MarchingSampling-Based Method for Optimal Motion Planning in Many Dimensions

Exercise

- https://bitbucket.org/castacks/core_planning_tutorial
- Fix the four steps in planning_tutorial.cpp (20 minute)
- Then follow the exercises outlined in the readme