Motion planning III: sampling-based planners

Vojtěch Vonásek

Department of Cybernetics
Faculty of Electrical Engineering
Czech Technical University in Prague

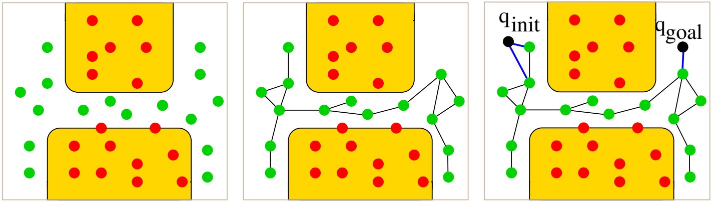
Summary from last lecture

Sampling-based planning

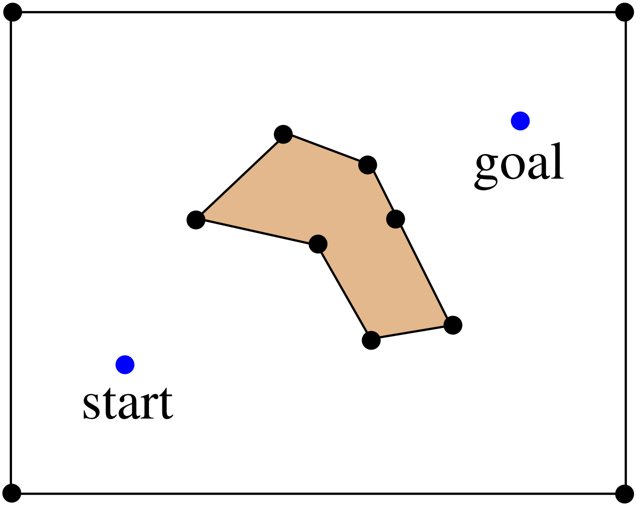
✓ Robots of arbitrary shapes
  - Robot shape is considered in collision detection
  - Collision detection is used as a "black-box"
  - Single-body or multi-body robots are allowed

✓ Robots with many-DOFs
  - Because the search is realized directly in $\mathcal{C}$-space
  - Dimension of $\mathcal{C}$ is determined by the DOFs

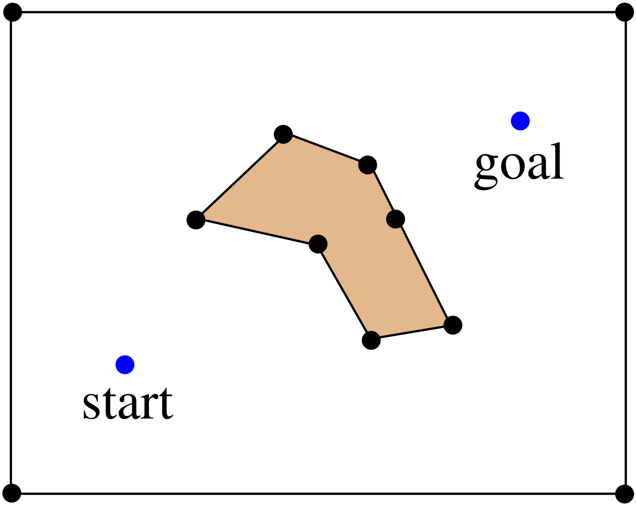
✓ Kinematic, dynamic and task constraints can be considered
  - It depends on the employed local planner



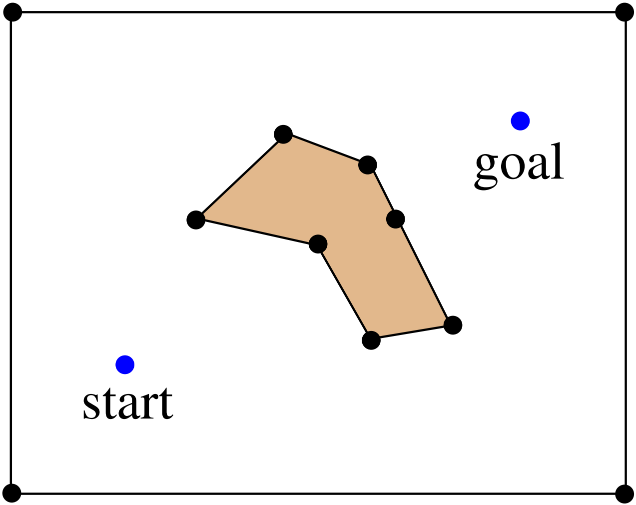
Draw Visibility graph + path from start to goal



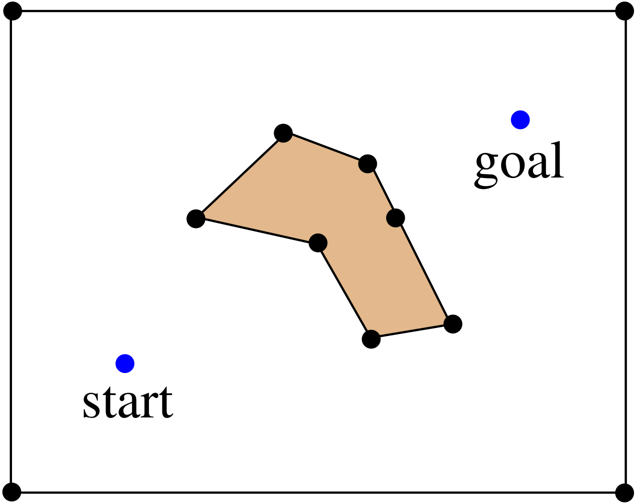
Draw Horizontal cell decomposition + path from start to goal



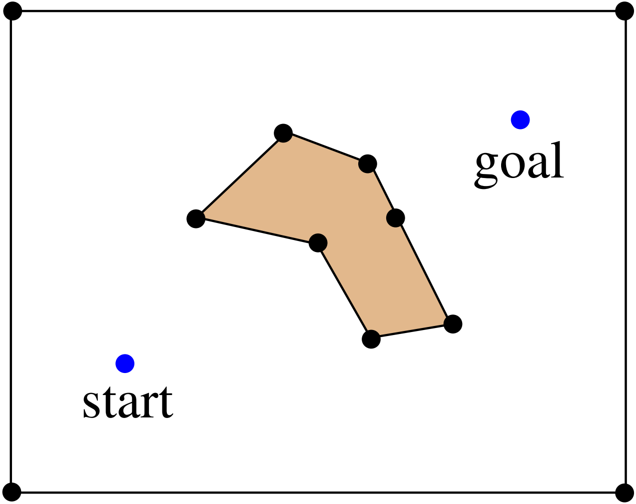
Draw Visibility graph for circle robot of radius *r* + path from start to goal



Draw PRM



Draw RRT



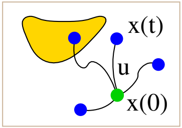
Lecture outline

- Examples of using RRT
  - For robotic manipulators
  - For car-like vehicles with Dubins maneuvers
  - For general simulated system
- Performance analysis
- Issues of sampling-based planning
- Basic modifications of RRT and PRM

Considering differential constraints

- Let assume the transition equation

$$\dot{x} = f(x, u)$$



where $x \in \mathcal{X}$ is a state vector and $u \in \mathcal{U}$ is an action vector from action space $\mathcal{U}$

- $\mathcal{X}$ is a state space, which may be $\mathcal{X} = \mathcal{C}$ or a phase space
  - Phase space is derived from $\mathcal{C}$ if dynamics is considered
  - Similarly to $\mathcal{C}$, $\mathcal{X}$ has $\mathcal{X}_{\mathrm{free}}$ and $\mathcal{X}_{\mathrm{obs}}$
- $f(x, u)$ is also called **forward motion model**
- Let $\tilde{u} : [0, \infty] \to \mathcal{U}$ is the action trajectory
- Action at time t is $\tilde{u}(t) \in \mathcal{U}$
- **State trajectory** is derived form $\tilde{u}(t)$ as

$$x(t) = x(0) + \int_0^t f(x(t'), \tilde{u}(t')\mathrm{d}t'$$

where $x(0)$ is the initial state at $t = 0$

Planning under differential constraints

- Assume we have: world $\mathcal{W}$, robot $\mathcal{A}$, configuration space $\mathcal{C}$, state-space $\mathcal{X}$ and action space $\mathcal{U}$, start and goal states $x_{\mathrm{init}}, x_{\mathrm{goal}} \in \mathcal{X}_{\mathrm{free}}$
- A system specified using $\dot{x} = f(x, u)$
- The task is to compute the action trajectory $\tilde{u} : [0, \infty] \to \mathcal{U}$ that satisfies: $x(0) = x_{\mathrm{init}}$, $x(t) = x_{\mathrm{goal}}$ for some $t > 0$, $x(t) \in \mathcal{X}_{\mathrm{free}}$, where $x(t)$ is given by

$$x(t) = x(0) + \int_0^t f(x(t'), \tilde{u}(t')\mathrm{d}t'$$

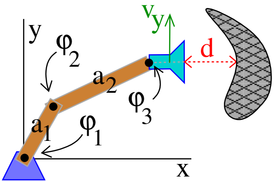
- This defines general motion planning under differential constraints

FACULTY OF ELECTRICAL ENGINEERING CTU IN PRAGUE | MRS MULTI-ROBOT SYSTEMS GROUP

Types of differential constraints

- Kinematics, usually given by motion model $\dot{x} = f(x, u)$
- Dynamics, e.g. $|\dot{x_6}| < x_{6,max}$ (e.g. to limit speed/acceleration)
- Task constraints, e.g. $\pi - \epsilon \leq x_{eff} \leq \pi + \epsilon$, where x_{eff} is the rotation of robotic arm effector

Example: robot measures an object using a sensor

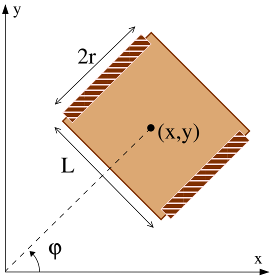


- How end-effector moves depending on $\varphi_1, \varphi_2, \varphi_3$ (transformation matrices) $\rightarrow$ kinematics constraints
- The sensor cannot move faster than v_y — dynamic constraint
- The sensor must be at distance d from the object — task constraint

Useful motion models

- Differential drive: control inputs are speeds of left/right wheel (u_l and u_r)

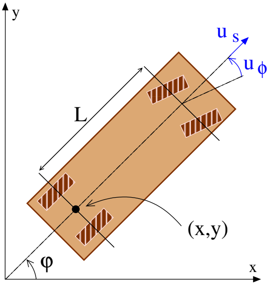
$$\begin{aligned}\dot{x} &= \frac{r}{2}(u_l + u_r)\cos\varphi \\ \dot{y} &= \frac{r}{2}(u_l + u_r)\sin\varphi \\ \dot{\varphi} &= \frac{r}{L}(u_r - u_l)\end{aligned}$$



Differential drive

- Car-like: control inputs are forward velocity u_s and steering angle u_ϕ

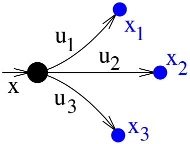
$$\begin{aligned}\dot{x} &= u_s\cos\varphi \\ \dot{y} &= u_s\sin\varphi \\ \dot{\varphi} &= \frac{u_s}{L}\tan u_\phi\end{aligned}$$



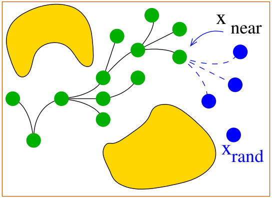
Car-like

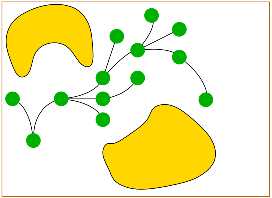
RRT for planning under diff. constr

- Similar to basic RRT
- Expansion of the tree using motion model and discretized input set $\mathcal{U}$

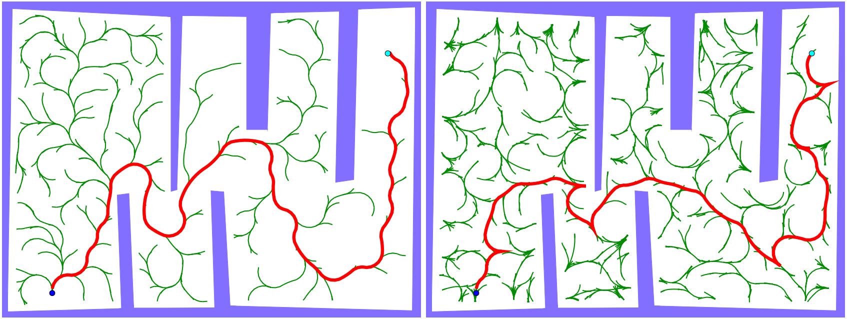


initialize tree $\mathcal{T}$ with x_{init}
for $i = 1, \ldots, I_{max}$ **do**
    x_{rand} = generate randomly in $\mathcal{X}$
    x_{near} = find nearest node in $\mathcal{T}$ towards x_{rand}
    $best = \infty$
    $x_{\text{new}} = \emptyset$
    foreach $u \in \mathcal{U}$ **do**
        x = integrate $f(x, u)$ from x_{near} over time Δt
        if *x is feasible* **and** *x is collision-free* **and** $\varrho(x, x_{\text{rand}}) < best$ **then**
            $x_{\text{new}} = x$
            $best = \varrho(x, x_{\text{rand}})$
    if $x_{\text{new}} \neq \emptyset$ **then**
        $\mathcal{T}$.addNode(x_{new})
        $\mathcal{T}$.addEdge($x_{\text{near}}, x_{\text{new}}$)
        if $\varrho(x_{\text{new}}, x_{\text{goal}}) < d_{goal}$ **then**
            return path from x_{init} to x_{goal}





RRT: example with car-like robot



Car-like, forward only

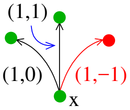
Car-like forward+backward motion

Enabling/disabling backward motion of car-like

- Either by assuming $u_s \geq 0$ (for forward motion only)
- Or explicit validation of results from local planner

  line 9: if x is feasible

- We have a car-like robot with broken steering mechanisms
- The robot can go either forward-only, or forward-and-left only
- Since robot is 2D and translation+rotation is required: $\mathcal{C}$ is 3D
- State space: $\mathcal{X} = \mathcal{C}$



$$\dot{x} = u_s \cos\varphi \quad \dot{y} = u_s \sin\varphi \quad \dot{\varphi} = \frac{u_s}{L} \tan u_\phi$$

$$\dot{\varphi} \geq 0$$

Practical implementation

- Determine action variables:

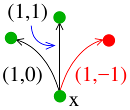
$$u_{s,min} \leq u_s \leq u_{s,max}$$

$$u_{\phi,min} \leq u_\phi \leq u_{\phi,max}$$

- Discretize each range, e.g. to m values $\rightarrow m^2$ combinations of $u_s \times u_\phi$
- For example: $\mathcal{U} = \{(-1,-1),(-1,0),(-1,1),(0,-1),(0,1),\ldots,(1,1)\}$
- Apply all $u \in \mathcal{U}$ during tree expansion, cut off infeasible states

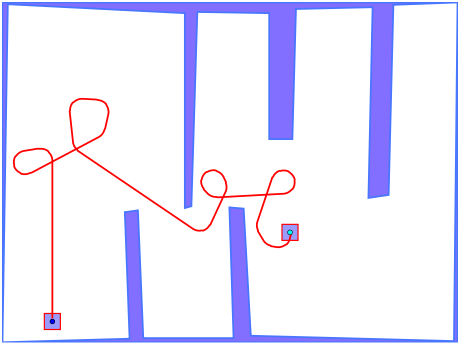
Example of RRT under diff. constraints

- We have a car-like robot with broken steering mechanisms
- The robot can go either forward-only, or forward-and-left only
- Since robot is 2D and translation+rotation is required: $\mathcal{C}$ is 3D
- State space: $\mathcal{X} = \mathcal{C}$



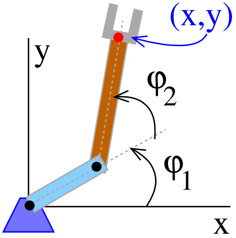
$$\dot{x} = u_s \cos\varphi \quad \dot{y} = u_s \sin\varphi \quad \dot{\varphi} = \frac{u_s}{L} \tan u_\phi$$

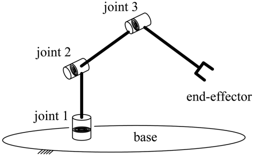
$$\dot{\varphi} \geq 0$$



RRT for manipulators I

- $q = (\varphi_1, \ldots, \varphi_n)$, n joints
- x = position of the link/end-effector
- x can contain also rotation if needed
- Forward kinematics: $x = FK(q)$
- Inverse kinematics: $q = IK(x)$
- Collision detection needs joint coordinates!
  - We need $\mathcal{A}_i(q)$ (position of link i at q)
  - Collision detection is between $\mathcal{A}_i(q)$ and $\mathcal{O}$
- Collision detection for end-effector pose x:
  - Compute $q = IK(x)$
  - Derive $A_i(q)$





Two arms
links $\mathcal{A}_1$ and $\mathcal{A}_2$

Spaces:

- Workspace/Cartesian space/Operation space — we plan path for end-effector (IK to joint space)
- Joint-space — we plan path by driving joints (FK to end-effector)

RRT for manipulators II

Planning via inverse kinematics

- We plan path of end-effector in workspace
- Naïve usage of RRT for manipulators
- Sampling, tree growth, nearest-neighbor s. in $\mathcal{W}$
- x_{rand} is generated randomly from $\mathcal{W}$

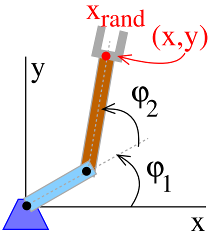
→ x_{rand} is the position of end-effector!

- x_{near} nearest in tree towards x_{rand}
- Make straigh-line from x_{near} to x_{rand} with resolution ε
- For each waypoint x on the line:
  - $q = IK(x)$, check collisions at q

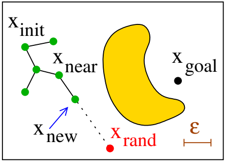
✗ Problem with singularities
  - line from x_{near} to x_{rand} may contain singularity
  - it may result in unwanted reconfiguration

✗ Requires (fast) inverse kinematics

✗ Task/dynamic constraints difficult to evaluate



$x = (x, y) \in \mathcal{W}$



tree is in $\mathcal{W}$

RRT for manipulators III

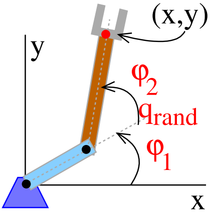
Planning via forward kinematics

- We plan path in joint-space (=$\mathcal{C}$)
- Sampling, tree growth and nearest-neighbor s. in $\mathcal{C}$
- Assume that joint i can change by $\pm\Delta_i$
- $\mathcal{U}$ is set of possible changes of the joints, e.g.:
  $\mathcal{U} = \{(-\Delta_1, 0), (\Delta_1, 0), (0, -\Delta_2), (0, \Delta_2), \dots\}$
- q_{rand} is generated randomly in $\mathcal{C}$
- q_{near} is its nearest neighbor in $\mathcal{T}$
- Tree expansion: for each $u \in \mathcal{U}$:
  - Apply u to q_{near}: $q' = q_{\text{near}} + u$
  - Check collision of $A_i(q')$
  - add to tree such q' that is collision-free and minimizes distance to q_{rand}

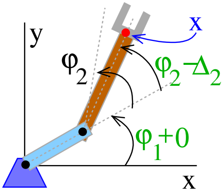
✗ Goal state needs to be defined in $\mathcal{C}$!

✓ No issues with singularities

✓ Task/dynamics constraints can be easily checked



$q = (\varphi_1, \varphi_2) \in C$
tree is in $\mathcal{C}$



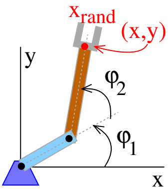
$u = (0, -\Delta_2)$,
$q' = (\varphi_1 + 0, \varphi_2 - \Delta_2)$

RRT for manipulators IV

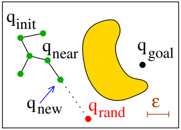
Planning with the task-space bias

- Combination of the two previous approaches
- Sampling in $\mathcal{W}$ (task-space), tree growth in $\mathcal{C}$ (joint space)
- Each node in the tree is (q, x), $q \in \mathcal{C}$, $x \in \mathcal{W}$
  - q-part is used for the tree expansion
  - x-part is used for the nearest-neighbor search
- x_{rand} is generated randomly from $\mathcal{W}$,
- x_{near} is nearest node from $\mathcal{T}$ towards x_{rand} measured in $\mathcal{W}$
- Get joint angles: $q_{\text{rand}} = IK(x_{\text{rand}})$ and $q_{\text{near}} = IK(x_{\text{near}})$
- q_{new} = straight-line expansion from q_{near} to q_{rand} (in $\mathcal{C}$)
- add q_{new} and $FK(q_{\text{new}})$ to the tree if it's collision-free

✓ Advantages: no problem with singularities, can handle task/dynamic constraints, the goal can be specified only in task space



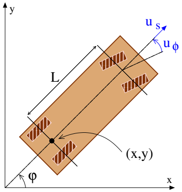
$q = (\varphi_1, \varphi_2)$
$\mathcal{C}$ is 2D



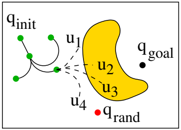
Local planner: Dubins curves

- Let's assume a simplified Car-like car moving by a constant forward speed $u_s = 1$:

$$\begin{aligned} \dot{x} &= \cos\varphi \\ \dot{y} &= \sin\varphi \\ \dot{\varphi} &= u \end{aligned}$$



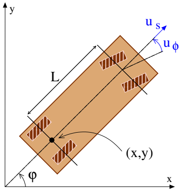
- control input (turning): $u = [-\tan\phi_{max}, \tan\phi_{max}]$



- Assume a RRT planner
- How to connect q_{near} to q_{rand}
- Naïve approach
  - try several u
  - use such u that minimizes distance to q_{rand}
- Or use Dubins vehicle!

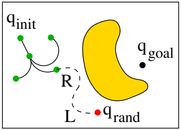
☛ L. E. Dubins, On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal position and tangents, American Journal of Mathematics, 79 (3): 497–516, 1957.

- Let's assume a simplified Car-like car moving by a constant forward speed $u_s = 1$:



$$\begin{aligned}\dot{x} &= \cos\varphi \\ \dot{y} &= \sin\varphi \\ \dot{\varphi} &= u\end{aligned}$$

- control input (turning): $u = [-\tan\phi_{max}, \tan\phi_{max}]$



Dubins curves

- Six optimal Dubins curves: LRL, RLR, LSL, LSR, RSL, RSR; S-straight, L-left, R-right
- Any two configurations can be optimally connected by these curves
- Useful as optimal "local-planner"

☛ L. E. Dubins, On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal position and tangents, American Journal of Mathematics, 79 (3): 497–516, 1957.

Random question

Is PRM better than RRT?

Performance measurement

Which planner is the best?

- Many planners, many modifications, many parameters
- No free lunch theorem!
- Selection of planner/parameters depends on the instance
- We cannot rely on literature/web
- Time complexity analysis does not always help
- We have to measure performance by ourself

Typical indicators:

- Path quality (length, time-to-travel, smoothness)
- Runtime & memory requirements
- Randomized planners: all above (statistically) + success rate curve

Good practice

- Testing setup should be as similar as possible to real situation
- Don't trust the test routine!, verify it first!!

Planner analysis: time complexity

- k is the number of collision detection queries
- $m_{\mathcal{A}}$ and $m_{\mathcal{W}}$ is the number of geometric objects describing $\mathcal{A}$ and $\mathcal{W}$
- *NN* is the complexity of nearest-neighbor search
- *CD* is the complexity of collision detection

initialize tree $\mathcal{T}$ with q_{init}
for $i = 1, \ldots, I_{max}$ **do**
    q_{rand} = generate randomly in $\mathcal{C}$
    q_{near} = nearest node in $\mathcal{T}$ towards q_{rand}
    q_{new} = localPlanner $q_{\text{near}} \rightarrow q_{\text{rand}}$
    if *canConnect*($q_{\text{near}}, q_{\text{new}}$) **then**
        $\mathcal{T}$.addNode(q_{new})
        $\mathcal{T}$.addEdge($q_{\text{near}}, q_{\text{new}}$)
        if $\varrho(q_{\text{new}}, q_{\text{goal}}) < d_{goal}$ **then**
            return path from q_{init} to q_{goal}

- Time complexity of one iteration of RRT with n nodes

$$\mathcal{O}(NN(n) + k \cdot CD(m_{\mathcal{A}}, m_{\mathcal{W}}))$$

- Assuming KD-tree for nearest-neighbor and hierarchical collision detection:

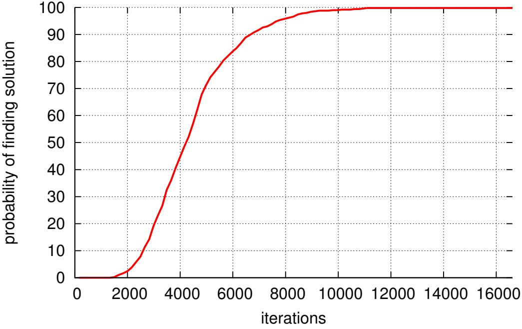
$$\mathcal{O}(\log n + k \log(m_{\mathcal{A}} + m_{\mathcal{W}}))$$

- General approach, valid for all methods

Planner analysis: cumulative probability

- Cumulative distribution function $F(x)$
- x is usually number of iterations (or runtime)

$\rightarrow$ probability that a plan is found in less than x iterations (or in time $< x$)

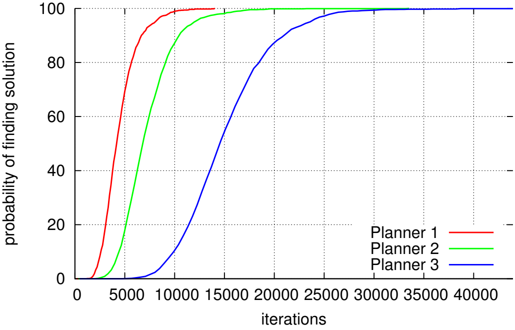


- For randomized planners only
- Results depend on tested scenario

Planner analysis: cumulative probability

- Cumulative distribution function $F(x)$
- x is usually number of iterations (or runtime)

→ probability that a plan is found in less than x iterations (or in time $< x$)



- For randomized planners only
- Results depend on tested scenario

Comparison of algorithms

We have two algorithms to use. How do we select better one?

Theorist

- We decide using complexity analysis $\mathcal{O}()\ldots$

Engineer

- We measure average runtime, memory, . . . , and see

Expert and student of ARO

- Not easy question, we need to consider:
  - What is the main criteria?
  - Range of scenarios/instances to be (typically) solved
  - Computational constraints (runtime limits, memory limits, . . .)
  - Robustness, implementation, dependencies

RRT vs Magic RRT: intro

Basic RRT

initialize tree $\mathcal{T}$ with q_{init}
for $i = 1, \ldots, I_{max}$ **do**
    q_{rand} = generate randomly in $\mathcal{C}$
    q_{near} = nearest node in $\mathcal{T}$ towards q_{rand}
    q_{new} = localPlanner $q_{\mathrm{near}} \rightarrow q_{\mathrm{rand}}$
    if *canConnect*($q_{\mathrm{near}}, q_{\mathrm{new}}$) **then**
        $\mathcal{T}$.addNode(q_{new})
        $\mathcal{T}$.addEdge($q_{\mathrm{near}}, q_{\mathrm{new}}$)
        if $\varrho(q_{\mathrm{new}}, q_{\mathrm{goal}}) < d_{goal}$ **then**
            return path from q_{init} to q_{goal}

$$\mathcal{O}(\log n + k \log(m_{\mathcal{A}} + m_{\mathcal{W}}))$$

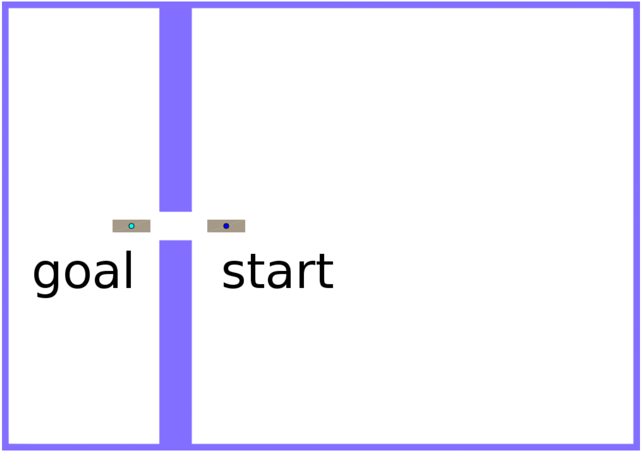
Magic RRT

initialize tree $\mathcal{T}$ with q_{init}
for $i = 1, \ldots, I_{max}$ **do**
    q_{rand} = generate randomly in $\mathcal{C}$
    if $i < 3$ **then**
        $q_{\mathrm{rand}} = q_{\mathrm{goal}}$
    q_{near} = nearest node in $\mathcal{T}$ towards q_{rand}
    q_{new} = localPlanner $q_{\mathrm{near}} \rightarrow q_{\mathrm{rand}}$
    if *canConnect*($q_{\mathrm{near}}, q_{\mathrm{new}}$) **then**
        $\mathcal{T}$.addNode(q_{new})
        $\mathcal{T}$.addEdge($q_{\mathrm{near}}, q_{\mathrm{new}}$)
        if $\varrho(q_{\mathrm{new}}, q_{\mathrm{goal}}) < d_{goal}$ **then**
            return path from q_{init} to q_{goal}

$$\mathcal{O}(\log n + k \log(m_{\mathcal{A}} + m_{\mathcal{W}}))$$

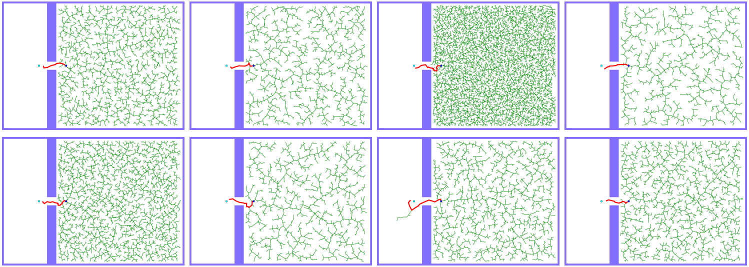
- Both methods have the same time complexity
- …but do they behave same?

RRT vs Magic RRT: scenario

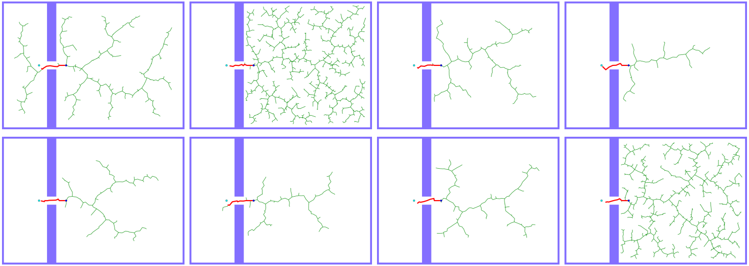


RRT vs Magic RRT: sample results

RRT, 8 trials

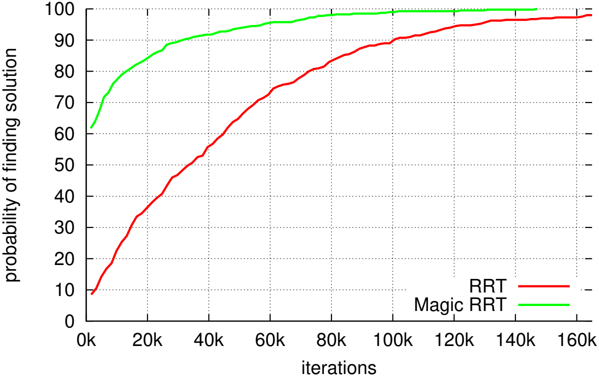


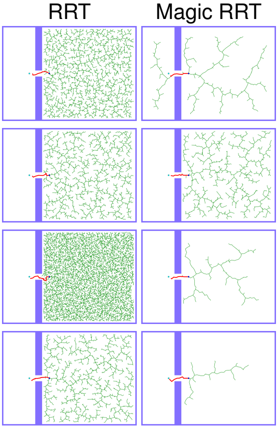
Magic RRT, 8 trials



- What is obvious difference between these two methods?

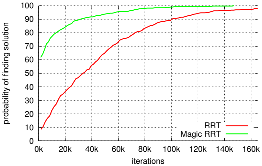
RRT vs Magic RRT: cum. probability

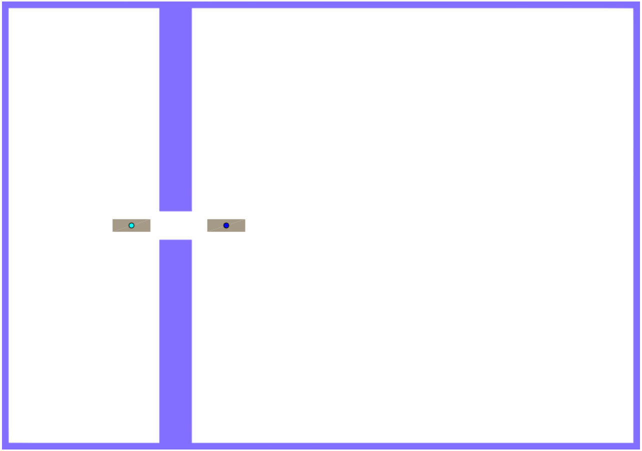




- Can you explain why Magic RRT is better?
- Is it true for all scenarios?
- Can you design a scenario where RRT will be better than Magic RRT?

RRT vs Magic RRT: cum. probability





RRT vs Magic RRT: conclusion

- In our scenario, RRT is worse than Magic RRT
- Above is true only for parameters used in the comparison!
- There are other scenarios with opposite behavior
- There are other scenarios where RRT is same (statistically) as Magic RRT
- Other parameters of RRT/Magic RRT, may lead to different results

Known issues of sampling-based planning

- One may consider sampling-based planning as a "magic" tool
  ...but that's not true at all!

Sampling-based planners have many issues

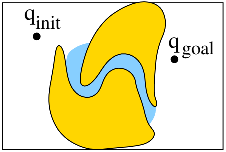
- Narrow passage problem
  - Difficulty of sampling small region in $\mathcal{C}_{\text{free}}$ surrounded by $\mathcal{C}_{\text{obs}}$
  - Problematic if (all) solutions have to pass that region
- Sensitivity to metric & parameters
  - How to measure distance in $\mathcal{C}$?
  - Selecting a good metric is as difficult as motion planning!
  - Many methods have "too many" parameters
  - Some parameters are hidden (or not well described)
  - How to tune the parameters?
- Supporting functions
  - Collision detection & nearest-neighbor search
  - Fast and reliable implementation

How do we recognize the issue? $\rightarrow$ performance measurement!

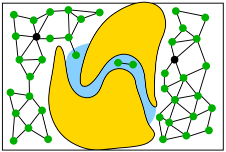
Narrow passage problem

Narrow passage (NP)

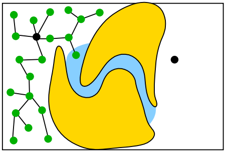
- A region $\mathcal{R} \subseteq \mathcal{C}_{\text{free}}$ with a small volume $vol(\mathcal{R}) < vol(\mathcal{C})$
- Probability that a random sample falls to $\mathcal{R}$ is $\sim vol(\mathcal{R})/vol(\mathcal{C})$
- NP are problematic if their removal changes connectivity of $\mathcal{C}_{\text{free}}$
- NP are regions in $\mathcal{C} \rightarrow$ they are given implicitly
- Location/size/volume/shape of NPs is not known!



narrow passage (NP)



PRM & NP

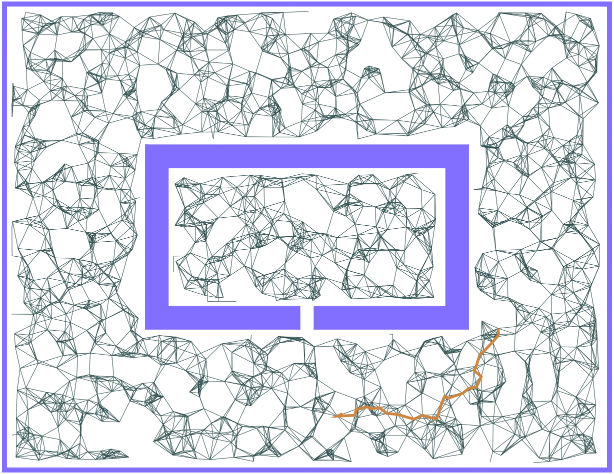


RRT/EST & NP

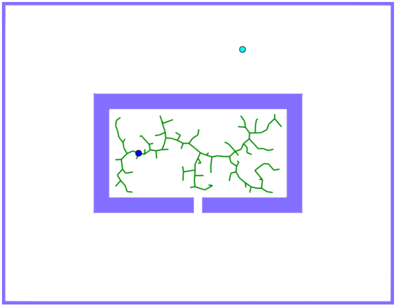
Consequences of having NP

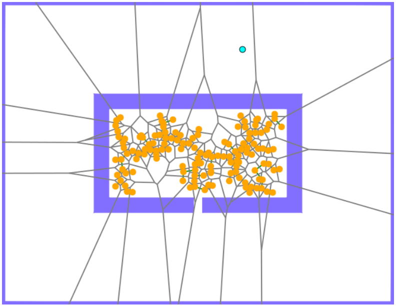
- PRM builds unconnected roadmaps $\rightarrow$ no solution
- RRT/EST cannot enter NP $\rightarrow$ no solution
- Number of samples must be significantly increased
- Runtime is increased

Narrow passage & PRM

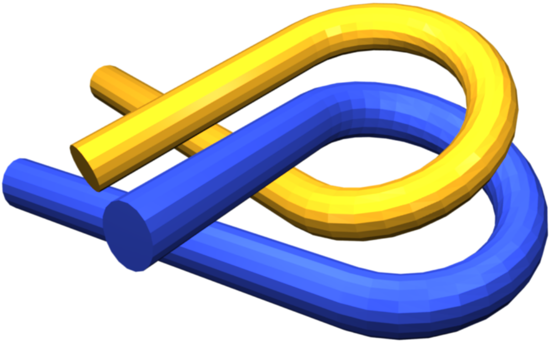


Narrow passage & RRT





Narrow passage



- Narrow passages are in $\mathcal{C}$
- Sometimes, we cannot (easily) see/estimate them from workspace!
- What makes the narrow passage in the Alpha-puzzle benchmark?