Motion planning
implementation details & data structures

Vojtěch Vonásek

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

Technical details

- Sampling-based planning relies on low-level routines
- Efficient implementation of these routines is necessary

- Random numbers generator
- Metric
- Nearest-neighbor search
- Collision-detection

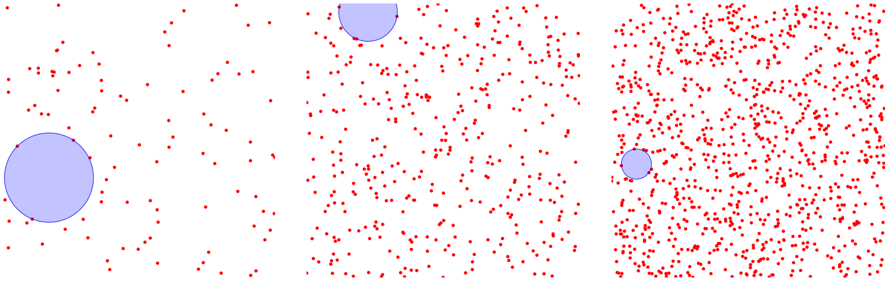
```
1  initialize tree $\mathcal{T}$ with $q_{\mathrm{init}}$
2  for $i = 1, \ldots, I_{max}$ do
3      $q_{\mathrm{rand}}$ = generate randomly in $\mathcal{C}$
4      $q_{\mathrm{near}}$ = find nearest node in $\mathcal{T}$ towards $q_{\mathrm{rand}}$
5      $q_{\mathrm{new}}$ = localPlanner from $q_{\mathrm{near}}$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

- These routines are required also in PRM, EST and all their variants

Generating random samples

Generation using standard `rand()`

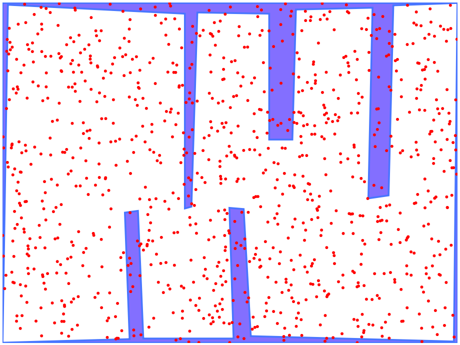
- Many variants of random number generator (RNG)
- RNG is (usually) implemented as a Linear Congruent Generator (LCG)
- Fast, easy for usage, provides "enough" number of samples
- High dispersion (the largest empty ball according to the used metric)


100 500 1000

Generating random samples

Generation using standard `rand()`

- Many variants of random number generator (RNG)
- RNG is (usually) implemented as a Linear Congruent Generator (LCG)
- Fast, easy for usage, provides “enough” number of samples
- High dispersion (the largest empty ball according to the used metric)

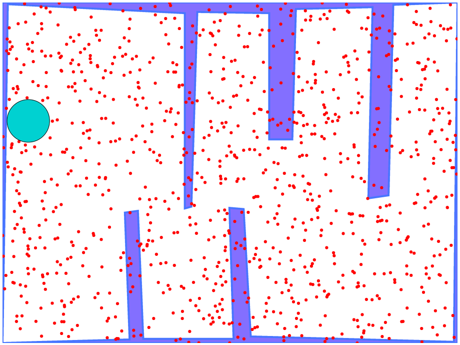


Random samples in 2D $\mathcal{C}$

Generating random samples

Generation using standard `rand()`

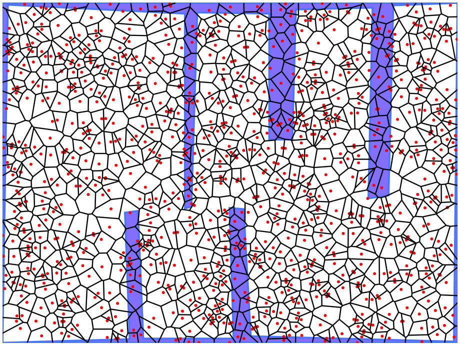
- Many variants of random number generator (RNG)
- RNG is (usually) implemented as a Linear Congruent Generator (LCG)
- Fast, easy for usage, provides “enough” number of samples
- High dispersion (the largest empty ball according to the used metric)


Random samples in 2D $\mathcal{C}$ + dispersion

Generating random samples

Generation using standard `rand()`

- Many variants of random number generator (RNG)
- RNG is (usually) implemented as a Linear Congruent Generator (LCG)
- Fast, easy for usage, provides "enough" number of samples
- High dispersion (the largest empty ball according to the used metric)



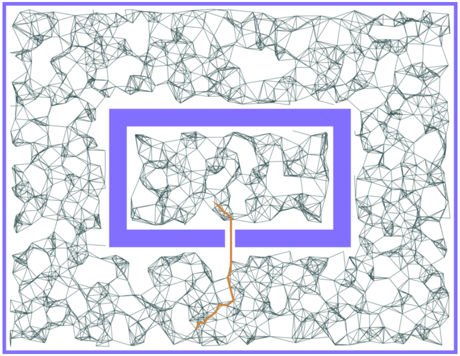
Voronoi diagram

Generating random samples

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

Generation using standard `rand()`

- Many variants of random number generator (RNG)
- RNG is (usually) implemented as a Linear Congruent Generator (LCG)
- Fast, easy for usage, provides "enough" number of samples
- High dispersion (the largest empty ball according to the used metric)

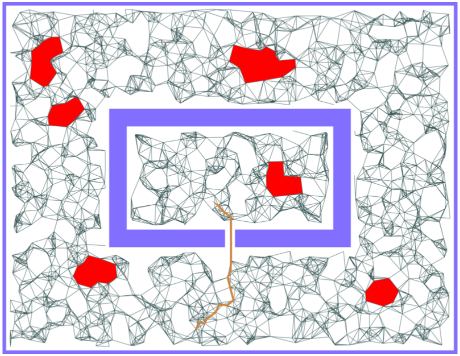


PRM roadmap, note the "holes"

Generating random samples

Generation using standard `rand()`

- Many variants of random number generator (RNG)
- RNG is (usually) implemented as a Linear Congruent Generator (LCG)
- Fast, easy for usage, provides “enough” number of samples
- High dispersion (the largest empty ball according to the used metric)



PRM roadmap, note the “holes” $\rightarrow$ due to the dispersion

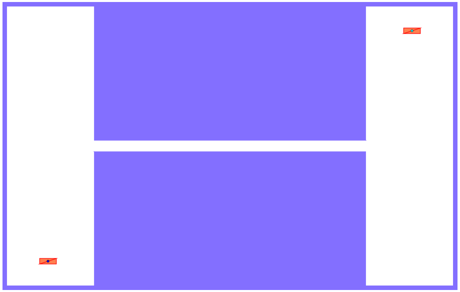
Generating random samples

Alternatives to `rand()`

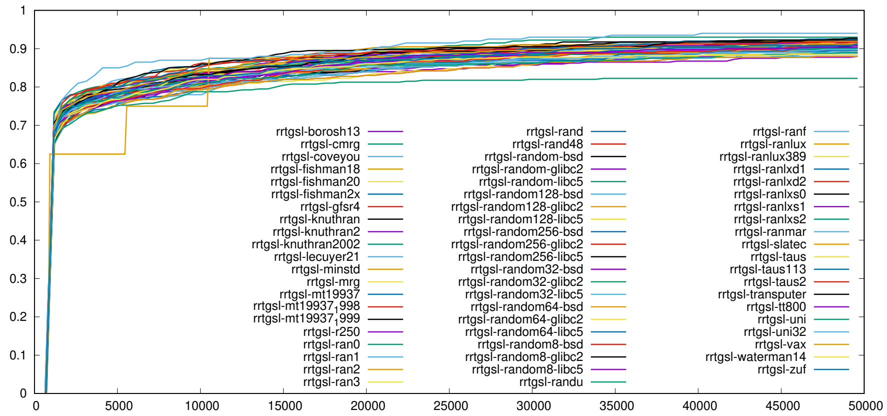
- Many libraries provide various RNG
    - e.g., Boost, GSL, numpy
- GSL — GNU Scientific library, offers tens of random generators
- Most of them are based on LCG

Does RNG influence the performance of sampling-based planners?

- Test scenario: square robot, 3D $\mathcal{C}$-space, narrow passage



Generating random samples

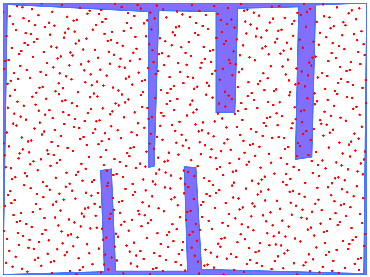


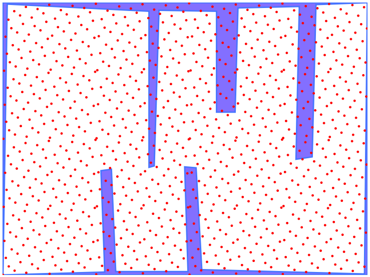
- What can you conclude from this measurement?

Generating random samples

Using low-discrepancy sequences

- Halton/Hammersley deterministic sequences
- Number of samples must be known in advance
- Slower computation in comparison to basic `rand()`
- Lower dispersion than LCG-based `rand()`


Halton points


Hammersley points

☛ J. M. Hammersley. Monte-Carlo methods for solving multivariable problems. Annals of the New York Academy of Science, 86:844–874, 1960.

Technical details

- Random numbers generator
- **Metric**
- Nearest-neighbor search
- Collision-detection

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

Metric I

- Sampling-based planners require a metric $\varrho(q_1, q_2)$, $q_1, q_2 \in \mathcal{C}$
- Often used are L_p metrics:

$$\varrho(x, x') = \left(\sum_{i=1}^{n} |x_i - x_i'|^p \right)^{1/p}$$

- L_2 is Euclidean metric
- L_1 is Manhattan metric
- Metric for 1D rotation:

$$\varrho(\theta_1, \theta_2) = \min\left(|\theta_1 - \theta_2|, 2\pi - |\theta_1 - \theta_2|\right)$$

- Metrics can be combined, let's assume that $\mathcal{C} = X \times Y$ with ϱ_X and ϱ_Y:

$$\varrho(q, q') = \left(c_x \varrho_X(x, x')^p + c_y \varrho_Y(y, y')^p \right)^{1/p}$$

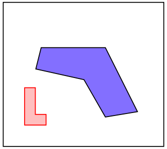
- where $c_x, c_y \geq 0$ are weights

Remind that for $a, b, c \in X$ in a metric space X and metric ϱ: $\varrho(a, b) \geq 0$; $\varrho(a, b) = 0$ if and only if $a = b$; $\varrho(a, b) = \varrho(b, a)$; $\varrho(a, b) + \varrho(b, c) \geq \varrho(a, c)$

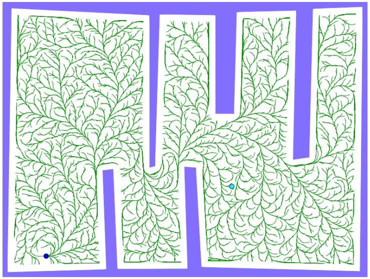
Metric II

- 2D object, translation + rotation $\rightarrow q = (x, y, \varphi) \in C$

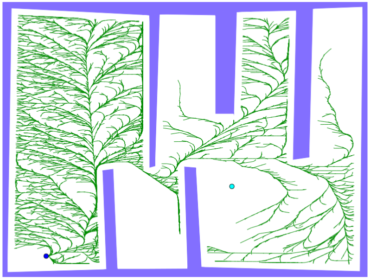
$$\varrho(q, q') = \sqrt{c_1((x - x')^2 + (y - y')^2) + c_2 \varrho_\theta(\varphi, \varphi')}$$



- where $\varrho_\theta(\varphi, \varphi')$ is the metric for 1D rotation
- The weights for translation c_1 is “usually” bigger than c_2, so the effect of the rotation is suppressed
- Wrong setting of weights can worse motion planning



Correct



Big weight on y- distance

Technical details

- Random numbers generator
- Metric
- **Nearest-neighbor search**
- Collision-detection

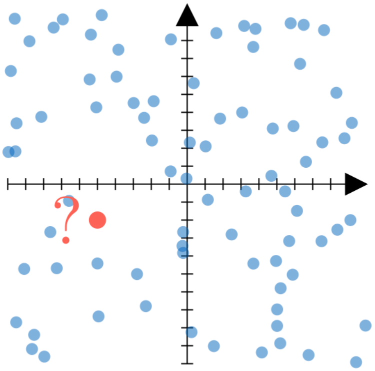
```
1  initialize tree $\mathcal{T}$ with $q_{\mathrm{init}}$
2  for $i = 1, \ldots, I_{max}$ do
3      $q_{\mathrm{rand}}$ = generate randomly in $\mathcal{C}$
4      $q_{\mathrm{near}}$ = find nearest node in $\mathcal{T}$ towards $q_{\mathrm{rand}}$
5      $q_{\mathrm{new}}$ = localPlanner from $q_{\mathrm{near}}$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

Nearest-neighbor search

- Given a set S, find a nearest point towards a query q
- Alternatives:
  - Find k nearest neighbors
  - Find all neighbors in the range r
- Naïve $\mathcal{O}(n)$ search is too slow!

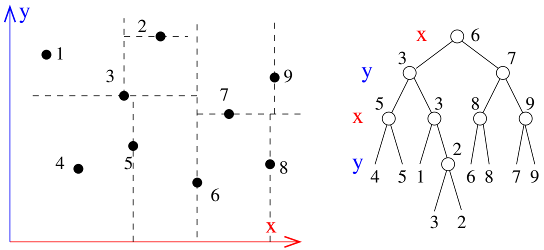
Challenges

- Fast query time
- Consider arbitrary metrics
- Dimensionality of S
- Fast preprocessing, low space requirements



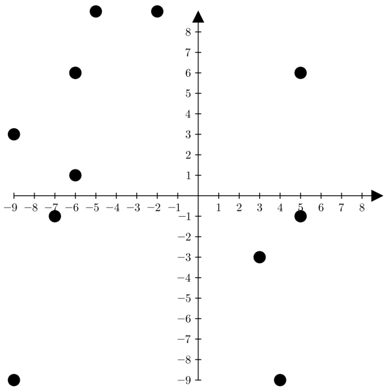
KD-trees: construction

- KD-tree is a binary tree, nodes represent a decision value
- Each level (of node) is for a different dimension



- Search is $\mathcal{O}(\log n)$ for n points in the KD-tree
- Construction $\mathcal{O}(dn \log n)$, n is number of points, d is dimension

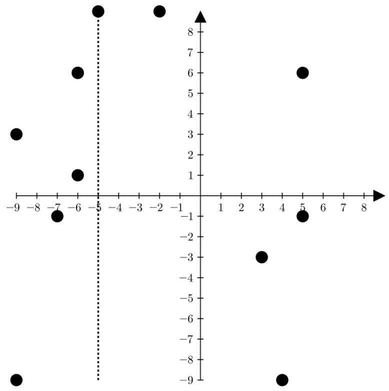
KD-Tree: construction



Construction of kd-trees

- Compute median in the given axis, make a new node (decision)
- Split points to two sets based on the decision
- Recursively build left and right subtrees, each subtree works with the next dimension

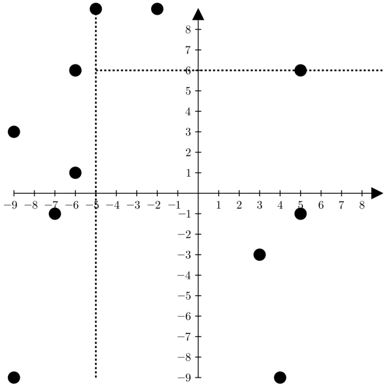
KD-Tree: construction

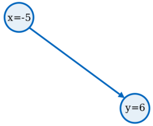


Construction of kd-trees

- Compute median in the given axis, make a new node (decision)
- Split points to two sets based on the decision
- Recursively build left and right subtrees, each subtree works with the next dimension

KD-Tree: construction

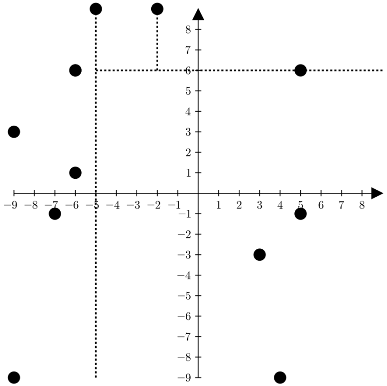


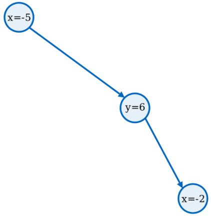


Construction of kd-trees

- Compute median in the given axis, make a new node (decision)
- Split points to two sets based on the decision
- Recursively build left and right subtrees, each subtree works with the next dimension

KD-Tree: construction

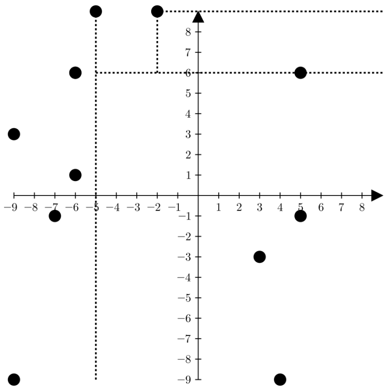


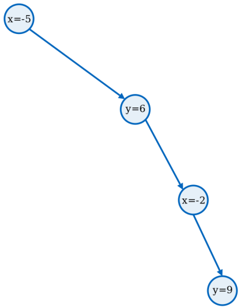


Construction of kd-trees

- Compute median in the given axis, make a new node (decision)
- Split points to two sets based on the decision
- Recursively build left and right subtrees, each subtree works with the next dimension

KD-Tree: construction

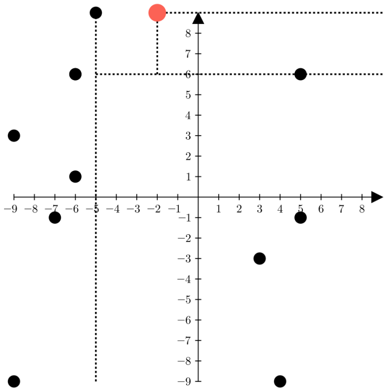


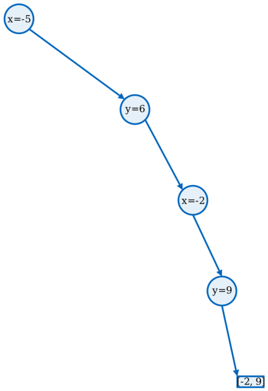


Construction of kd-trees

- Compute median in the given axis, make a new node (decision)
- Split points to two sets based on the decision
- Recursively build left and right subtrees, each subtree works with the next dimension

KD-Tree: construction

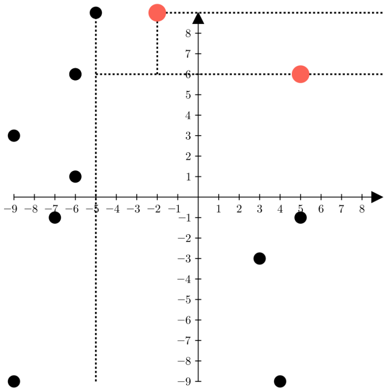


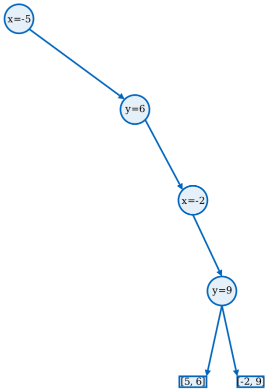


Construction of kd-trees

- Compute median in the given axis, make a new node (decision)
- Split points to two sets based on the decision
- Recursively build left and right subtrees, each subtree works with the next dimension

KD-Tree: construction

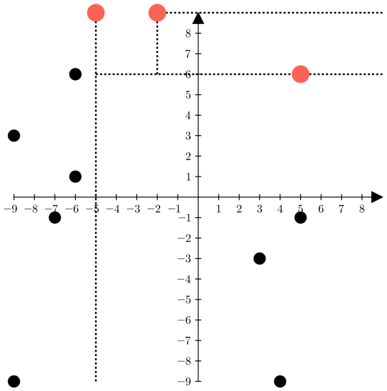


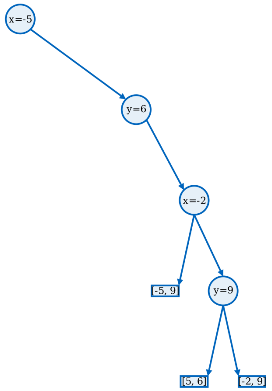


Construction of kd-trees

- Compute median in the given axis, make a new node (decision)
- Split points to two sets based on the decision
- Recursively build left and right subtrees, each subtree works with the next dimension

KD-Tree: construction

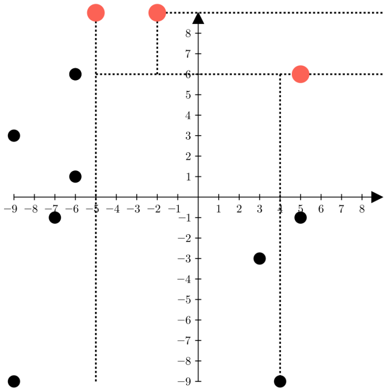


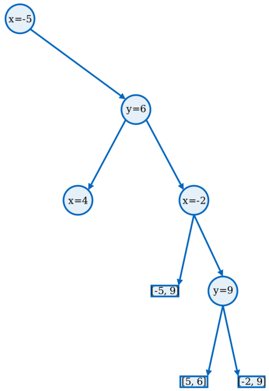


Construction of kd-trees

- Compute median in the given axis, make a new node (decision)
- Split points to two sets based on the decision
- Recursively build left and right subtrees, each subtree works with the next dimension

KD-Tree: construction

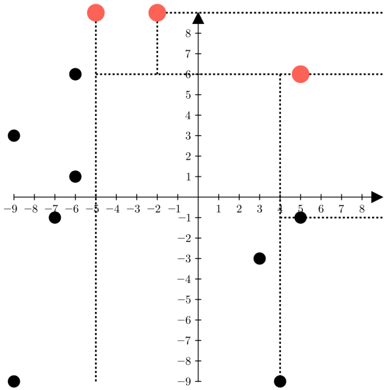


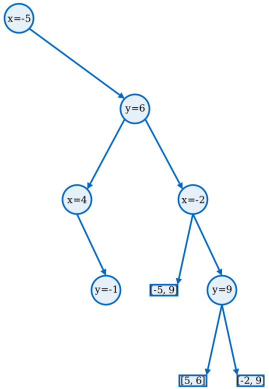


Construction of kd-trees

- Compute median in the given axis, make a new node (decision)
- Split points to two sets based on the decision
- Recursively build left and right subtrees, each subtree works with the next dimension

KD-Tree: construction

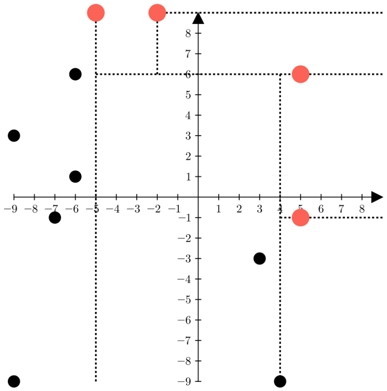


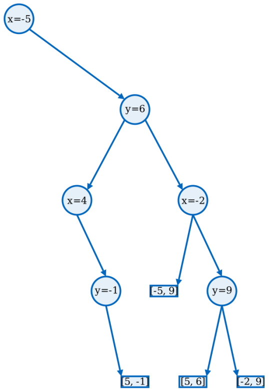


Construction of kd-trees

- Compute median in the given axis, make a new node (decision)
- Split points to two sets based on the decision
- Recursively build left and right subtrees, each subtree works with the next dimension

KD-Tree: construction

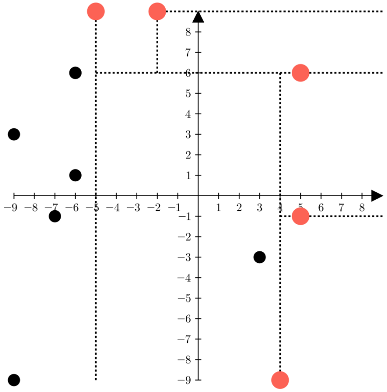


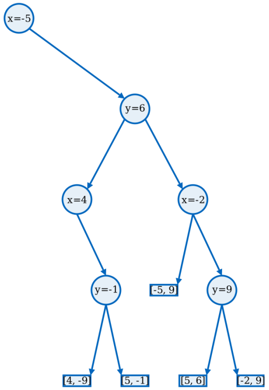


Construction of kd-trees

- Compute median in the given axis, make a new node (decision)
- Split points to two sets based on the decision
- Recursively build left and right subtrees, each subtree works with the next dimension

KD-Tree: construction

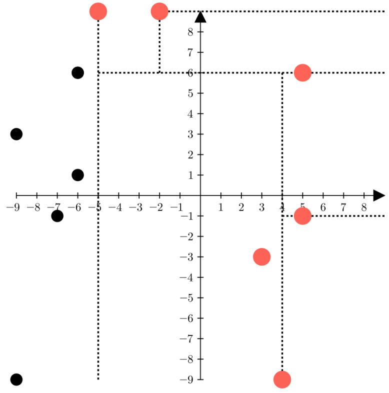


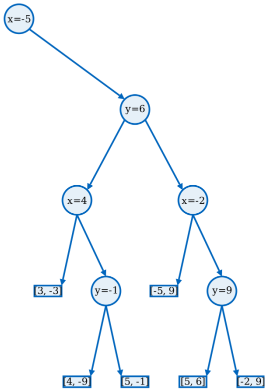


Construction of kd-trees

- Compute median in the given axis, make a new node (decision)
- Split points to two sets based on the decision
- Recursively build left and right subtrees, each subtree works with the next dimension

KD-Tree: construction

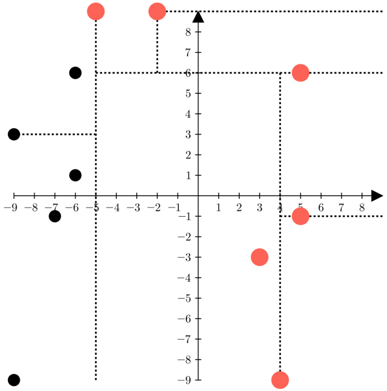


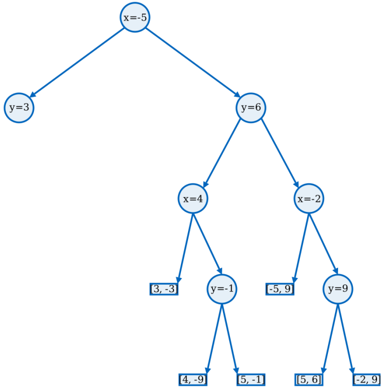


Construction of kd-trees

- Compute median in the given axis, make a new node (decision)
- Split points to two sets based on the decision
- Recursively build left and right subtrees, each subtree works with the next dimension

KD-Tree: construction

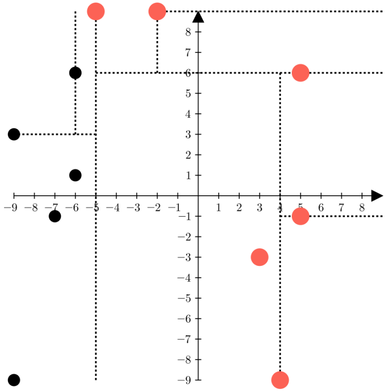


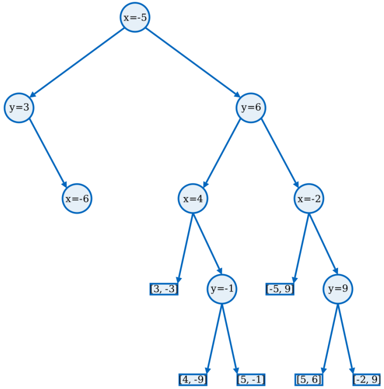


Construction of kd-trees

- Compute median in the given axis, make a new node (decision)
- Split points to two sets based on the decision
- Recursively build left and right subtrees, each subtree works with the next dimension

KD-Tree: construction

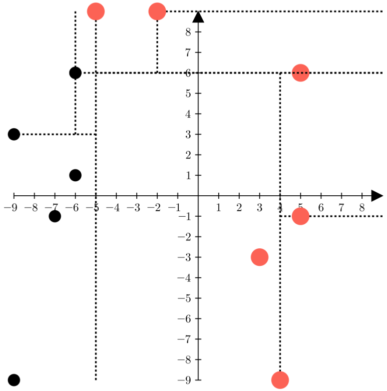


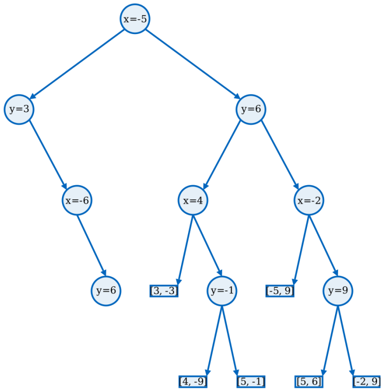


Construction of kd-trees

- Compute median in the given axis, make a new node (decision)
- Split points to two sets based on the decision
- Recursively build left and right subtrees, each subtree works with the next dimension

KD-Tree: construction

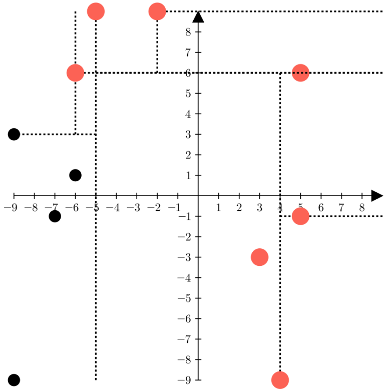


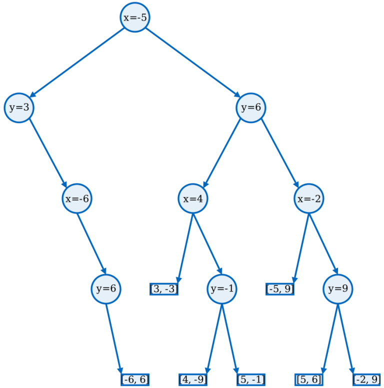


Construction of kd-trees

- Compute median in the given axis, make a new node (decision)
- Split points to two sets based on the decision
- Recursively build left and right subtrees, each subtree works with the next dimension

KD-Tree: construction

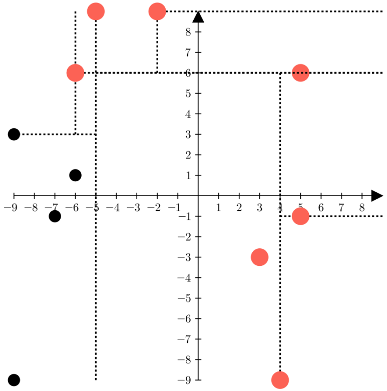


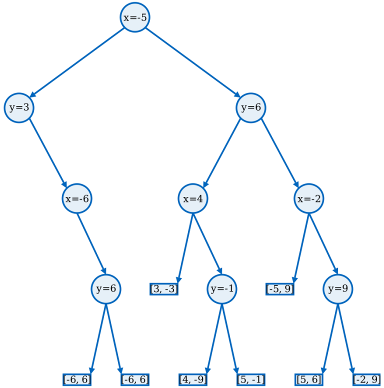


Construction of kd-trees

- Compute median in the given axis, make a new node (decision)
- Split points to two sets based on the decision
- Recursively build left and right subtrees, each subtree works with the next dimension

KD-Tree: construction

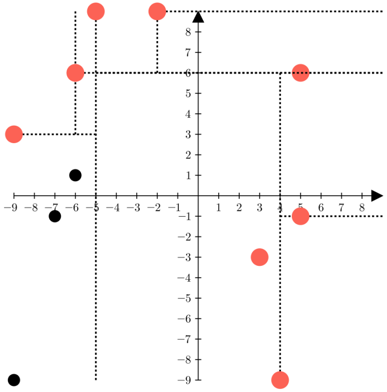


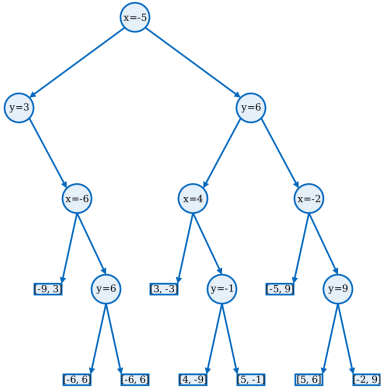


Construction of kd-trees

- Compute median in the given axis, make a new node (decision)
- Split points to two sets based on the decision
- Recursively build left and right subtrees, each subtree works with the next dimension

KD-Tree: construction

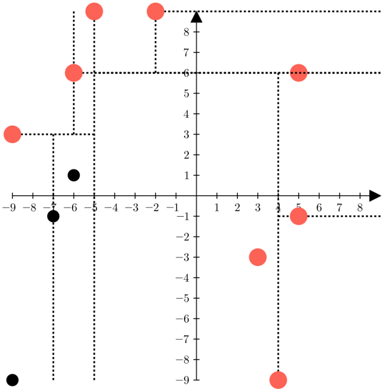


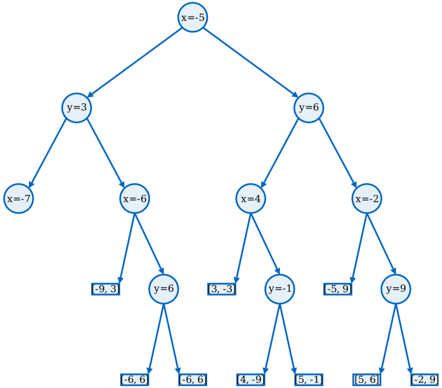


Construction of kd-trees

- Compute median in the given axis, make a new node (decision)
- Split points to two sets based on the decision
- Recursively build left and right subtrees, each subtree works with the next dimension

KD-Tree: construction

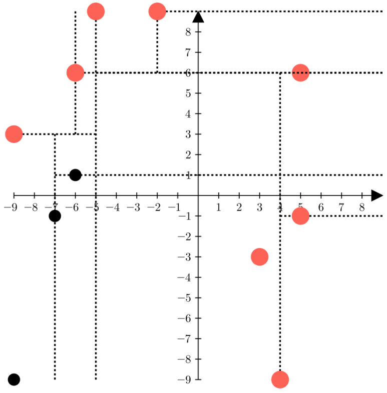


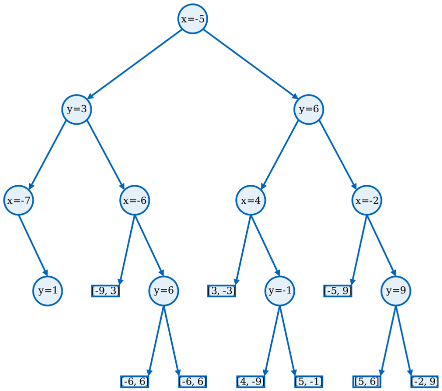


Construction of kd-trees

- Compute median in the given axis, make a new node (decision)
- Split points to two sets based on the decision
- Recursively build left and right subtrees, each subtree works with the next dimension

KD-Tree: construction

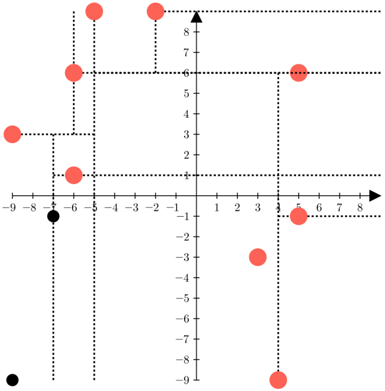


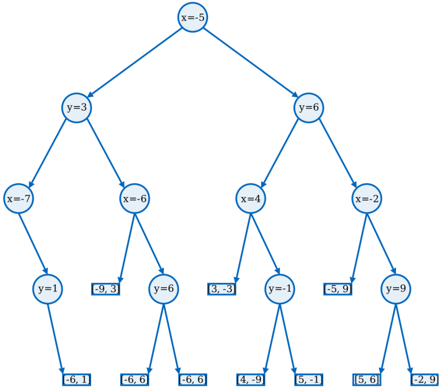


Construction of kd-trees

- Compute median in the given axis, make a new node (decision)
- Split points to two sets based on the decision
- Recursively build left and right subtrees, each subtree works with the next dimension

KD-Tree: construction

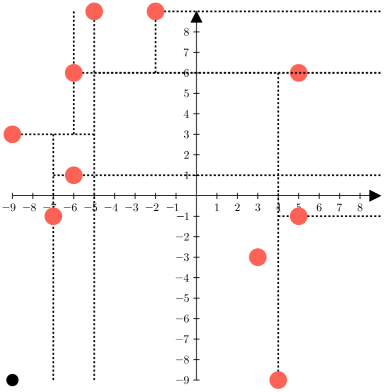


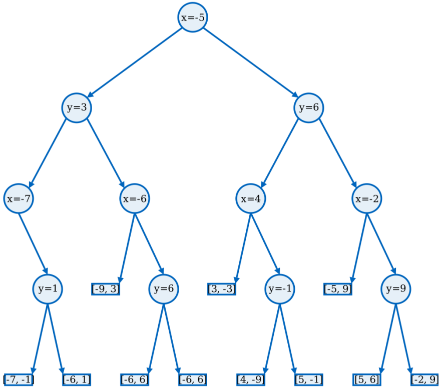


Construction of kd-trees

- Compute median in the given axis, make a new node (decision)
- Split points to two sets based on the decision
- Recursively build left and right subtrees, each subtree works with the next dimension

KD-Tree: construction

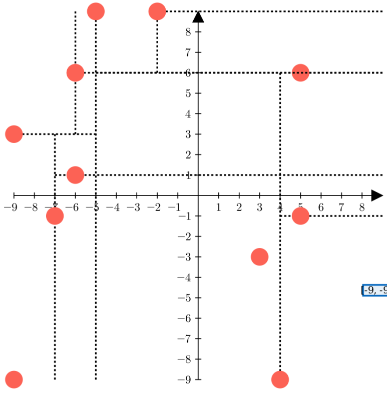
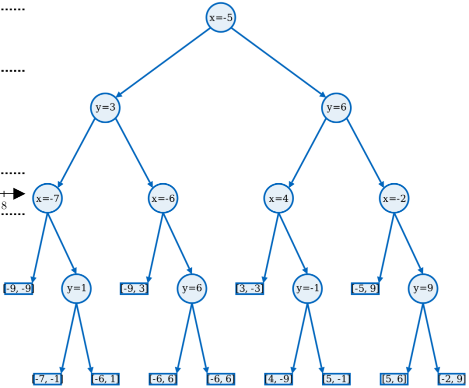




Construction of kd-trees

- Compute median in the given axis, make a new node (decision)
- Split points to two sets based on the decision
- Recursively build left and right subtrees, each subtree works with the next dimension

KD-Tree: construction

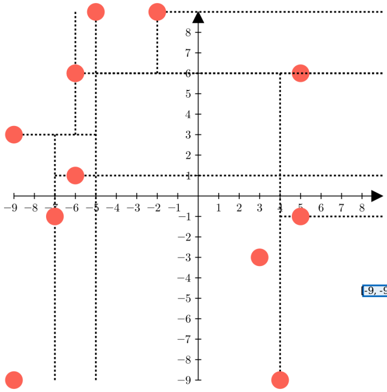
Construction of kd-trees

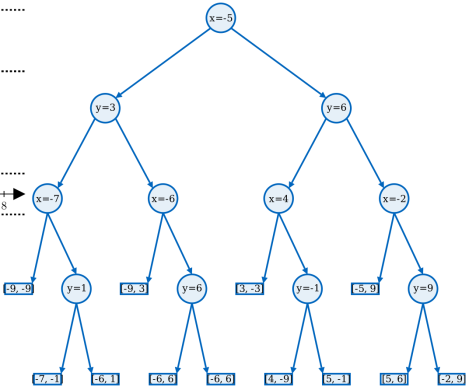
- Compute median in the given axis, make a new node (decision)
- Split points to two sets based on the decision
- Recursively build left and right subtrees, each subtree works with the next dimension

KD-Tree: search

Principle of nearest-neighbor search

- Input is a point
- Traverse the nodes till the leaf (based on decision in each node)
- This locates a region that **may** contain a nearest neighbor
- Search also all surrounding regions





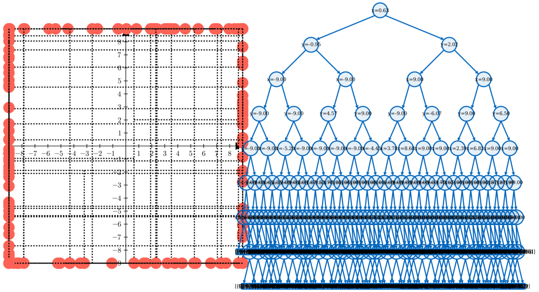
KD-Tree: issues

Usefull operations

- Inserting new item in $\mathcal{O}(\log n)$
- Removing existing item in $\mathcal{O}(\log n)$

KD-Tree issues

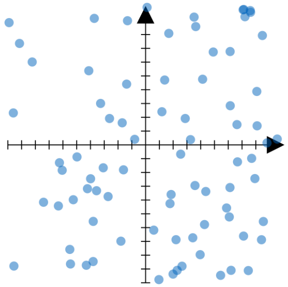
- Not suitable for other than Euclidean metrics
- Ineffective for large dimensions k
- It needs $n \gg 2^k$ data to achieve $\mathcal{O}(\log n)$ performance, otherwise it performs almost linear search
- Ineffective for non-uniform data



Geometric Nearest-neighbor Access Tree

Construction of GNAT

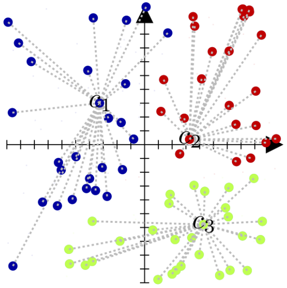
- Select m pivots $c_1, \dots c_m \in S$
- Assign each point in S to the nearest pivot, D_{c_i} (clusters)
- For each cluster D_{c_i}:
  - $R_{i,j} = [\min_{x \in X} \varrho(c_i, x), \max_{x \in X} \varrho(c_i, x)]$, $X = D_{c_j} \cup \{c_j\}$
  - $R_{i,j} = [\mathrm{low}, \mathrm{high}]$ are ranges of distances between c_i and data points of other clusters
- Build recursively the children c_i with its points D_{c_i}

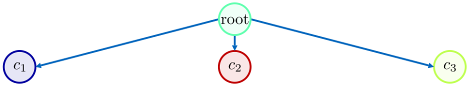


Geometric Nearest-neighbor Access Tree

Construction of GNAT

- Select m pivots $c_1, \dots c_m \in S$
- Assign each point in S to the nearest pivot, D_{c_i} (clusters)
- For each cluster D_{c_i}:
  - $R_{i,j} = [\min_{x \in X} \varrho(c_i, x), \max_{x \in X} \varrho(c_i, x)]$, $X = D_{c_j} \cup \{c_j\}$
  - $R_{i,j} = [\mathrm{low}, \mathrm{high}]$ are ranges of distances between c_i and data points of other clusters
- Build recursively the children c_i with its points D_{c_i}

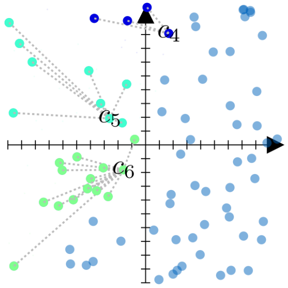


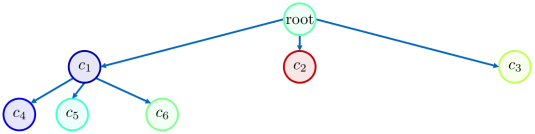


Geometric Nearest-neighbor Access Tree

Construction of GNAT

- Select m pivots $c_1, \dots c_m \in S$
- Assign each point in S to the nearest pivot, D_{c_i} (clusters)
- For each cluster D_{c_i}:
  - $R_{i,j} = [\min_{x \in X} \varrho(c_i, x), \max_{x \in X} \varrho(c_i, x)]$, $X = D_{c_j} \cup \{c_j\}$
  - $R_{i,j} = [\mathrm{low}, \mathrm{high}]$ are ranges of distances between c_i and data points of other clusters
- Build recursively the children c_i with its points D_{c_i}

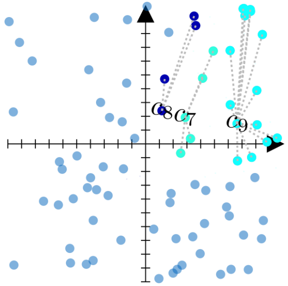


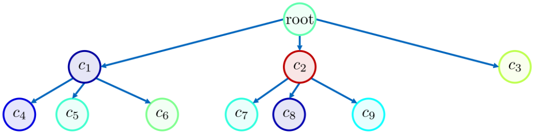


Geometric Nearest-neighbor Access Tree

Construction of GNAT

- Select m pivots $c_1, \dots c_m \in S$
- Assign each point in S to the nearest pivot, D_{c_i} (clusters)
- For each cluster D_{c_i}:
  - $R_{i,j} = [\min_{x \in X} \varrho(c_i, x), \max_{x \in X} \varrho(c_i, x)]$, $X = D_{c_j} \cup \{c_j\}$
  - $R_{i,j} = [\mathrm{low}, \mathrm{high}]$ are ranges of distances between c_i and data points of other clusters
- Build recursively the children c_i with its points D_{c_i}

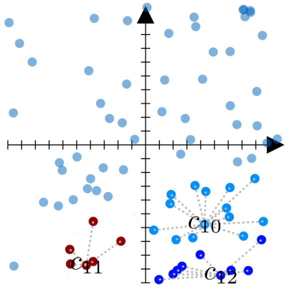


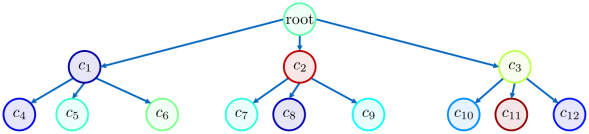


Geometric Nearest-neighbor Access Tree

Construction of GNAT

- Select m pivots $c_1, \dots c_m \in S$
- Assign each point in S to the nearest pivot, D_{c_i} (clusters)
- For each cluster D_{c_i}:
    - $R_{i,j} = [\min_{x \in X} \varrho(c_i, x), \max_{x \in X} \varrho(c_i, x)]$, $X = D_{c_j} \cup \{c_j\}$
    - $R_{i,j} = [\mathrm{low}, \mathrm{high}]$ are ranges of distances between c_i and data points of other clusters
- Build recursively the children c_i with its points D_{c_i}

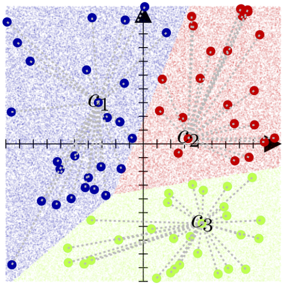


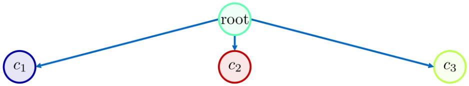


GNAT: search

Nearest neighbor search towards a query point q

1. Start at root
2. Select a pivot c_i
3. If $e = \varrho(q, c_i) \leq r$, report c_i
4. If $[e - r, e + r] \cap R_{i,j} = \emptyset$, we can prune node of cluster c_j
5. Repeat steps 2–4 for all clusters $i = 1, \ldots, m$ at the current level
6. For each non-pruned cluster c_i, search its corresponding subtree

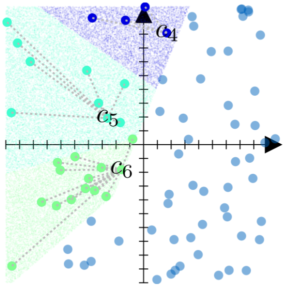


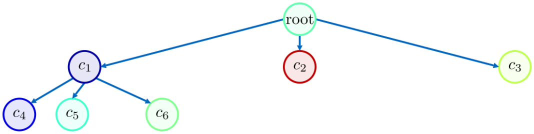


GNAT: search

Nearest neighbor search towards a query point q

1. Start at root
2. Select a pivot c_i
3. If $e = \varrho(q, c_i) \leq r$, report c_i
4. If $[e - r, e + r] \cap R_{i,j} = \emptyset$, we can prune node of cluster c_j
5. Repeat steps 2–4 for all clusters $i = 1, \dots, m$ at the current level
6. For each non-pruned cluster c_i, search its corresponding subtree

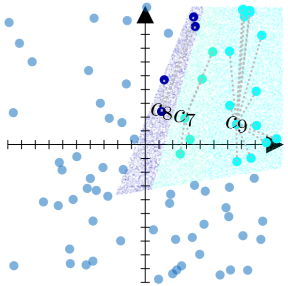


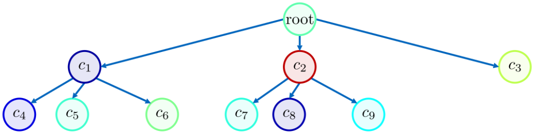


GNAT: search

Nearest neighbor search towards a query point q

1. Start at root
2. Select a pivot c_i
3. If $e = \varrho(q, c_i) \leq r$, report c_i
4. If $[e - r, e + r] \cap R_{i,j} = \emptyset$, we can prune node of cluster c_j
5. Repeat steps 2–4 for all clusters $i = 1, \ldots, m$ at the current level
6. For each non-pruned cluster c_i, search its corresponding subtree

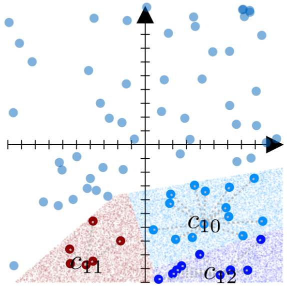


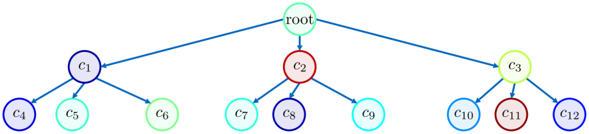


GNAT: search

Nearest neighbor search towards a query point q

1. Start at root
2. Select a pivot c_i
3. If $e = \varrho(q, c_i) \leq r$, report c_i
4. If $[e - r, e + r] \cap R_{i,j} = \emptyset$, we can prune node of cluster c_j
5. Repeat steps 2–4 for all clusters $i = 1, \ldots, m$ at the current level
6. For each non-pruned cluster c_i, search its corresponding subtree





GNAT: properties

Nearest neighbor search towards a query point q

- Assume m clusters at each node
- Construction (average) $\mathcal{O}(nm \log_m n)$, worst case $\mathcal{O}(n^2)$
- Space complexity $\mathcal{O}(mn)$
- Search: time complexity is hard to analyze, experiments show that it's $\sim$ logarithmic
- Practically, GNAT performs better for larger d than KD-trees
- GNAT works with arbitrary metric
- GNAT does not degenerate with non-uniform distributions

Technical details

- Random numbers generator
- Metric
- Nearest-neighbor search
- **Collision-detection**

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

Collision detection

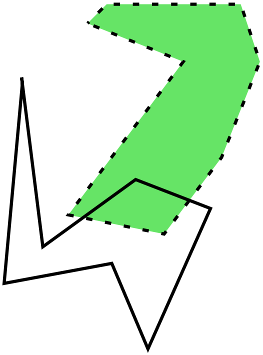
- Determines if/how objects collide/overlap/intersect/touch
- "Collision detection" covers two different techniques:

Collision-detection:

- True/False answer
- Fast, suitable for sampling-based planners

Collision-determination:

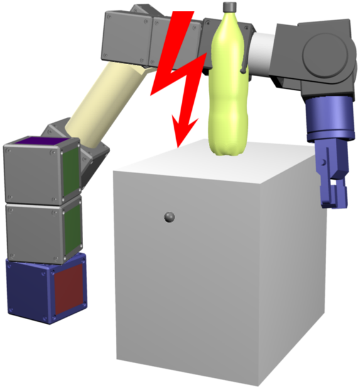
- Report details about collisions
- Identify which objects intersect
- Enumerate involved primitives
- Optionally computes the "penetration vector", or point of intersection
- Slower than collision-detection



Collision detection

Consider the manipulator at collision

- How can you react with collision-detection?
- How collision-determination helps to overcome the problem?



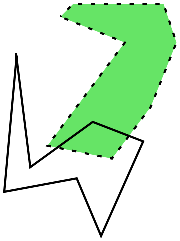
Collision detection (CD)

Geometric primitives

- Points, Lines, Circles, Triangles, Spheres, Cylinders, Rectangles
- Objects are constructed from these primitives
- The primitives determines which CD algorithm can be chosen
- CD relies on intersection tests between the primitives
- Convex shapes are always better, CD is faster with them

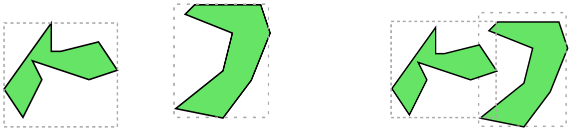
Collision detection between *n* and *m* primitives

- Naïve CD: $\mathcal{O}(mn)$
- This can be too slow!



Bounding volume

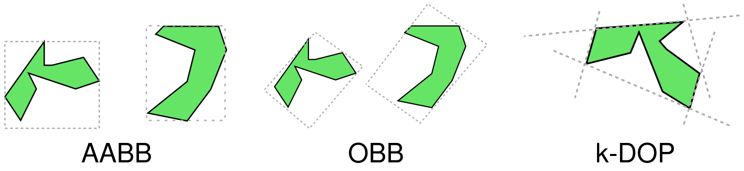
- Reduce complexity of CD by replacing the original object by a simpler object that contains the original one
- Represent an object by a Bounding Box (BB)
- If two BBs do not overlap, object inside cannot collide (fast test)
- If two BBs collide, further test is made using internal objects (slow test)



- BB should be geometrically simple to enable fast BB-vs-BB tests
- Spheres/circles, ellipses, rectangles
- BB should be as tight as possible to minimize false-positives

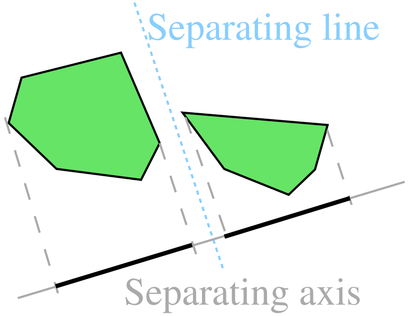
Rectangles as bounding boxes

- **AABB — Axis Aligend Bounding Box**
  - Faces of bounding box are parallel to the coordinated system
  - Very fast detection of overlap of two BBs
  - Not suitable for 'rotated' objects that lead to large BB
- **OBB — Oriented Bounding Box**
  - Faces of BB are oriented according to the object
  - Lower volume of BB, less false-positives
  - Slower detection of BBs overlap than for AABB
- **k-DOP — *k* Discrete Oriented Polytope**
  - Boolean intersection along *k* directions
  - Axes of DOPs do not have to be orthogonal
  - Generalization of AABB/OBB (e.g., AABB in 2D is 4-DOP)



Separation axis theorem

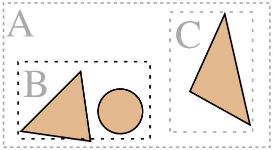
- Is used to determine overlap of two convex objects
- Two convex polytopes do not overlap if there exists a line onto which the projection of the two objects do not overlap
- Separating line can be determined by testing all combinations of lines/faces of both objects
- Convex objects!

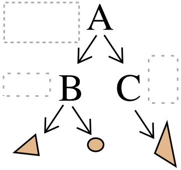


Bounding volume hierarchy

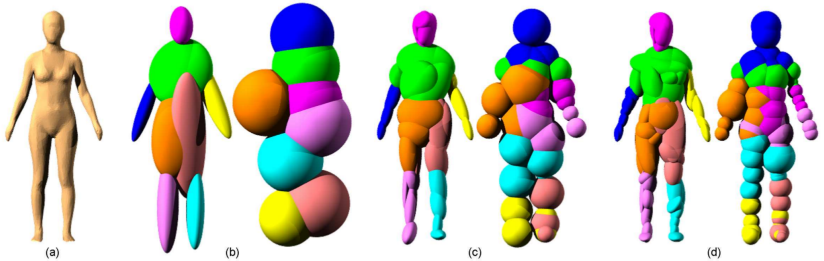
Bounding Volume Hierarchy (BVH)

- Original objects are recursively split to subsets
- BVH is a tree structure of bounding-boxes (BB) for each subset
- A Node in BVH is either a BB or a geometric object





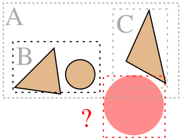
BVH using ellipsoids

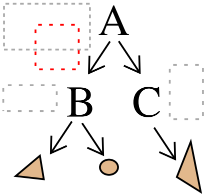


- S. Liu, C. C. L. Wang, K. Hui, X. Jin, H. Zhao. Ellipsoid-tree construction for solid objects. ACM symposium on Solid and physical modeling, 2007.

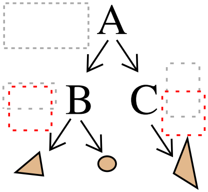
Collision detection using BVH

- **Broad phase**
  - Traverse BVH from the root
  - At each level, evaluate overlaps between BBs
  - If BBs do not overlap, return no-collision
  - If BBs overlap, continue to child nodes
- **Narrow phase**
  - for two overlapping BBs, perform collision detection of their internal objects
- Hierarchical CD: $\mathcal{O}(\log n)$ for n geometric primitives
- Building of BVH (depends on its type) takes at least $O(n)$

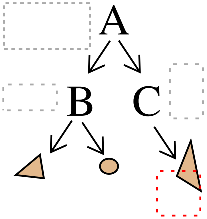




(BB vs BB)



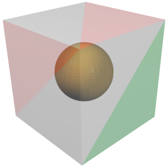
(BB vs BB)

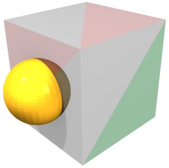


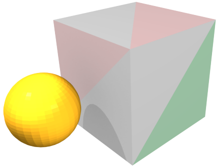
(Circle vs Triangle)

Notes on collision detection I

- Usual representation for 2D objects:
  - Combination of boxes/spheres, polygons, triangulated polygons
- Usual representation for 3D objects:
  - Combination of 3D geometric primitives (boxes,spheres,cylinders), triangle mesh
- Note: triangle meshes are hollow $\rightarrow$ detection of ‘object inside object’ is not possible

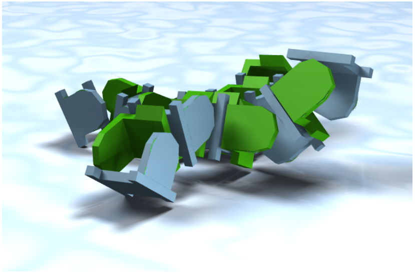

No collision

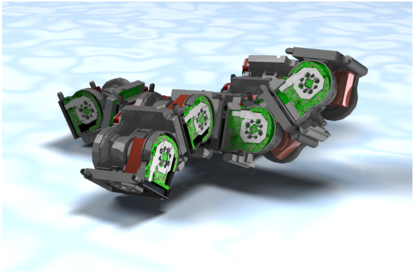

Collision


No collision

Notes on collision detection II

- CD between objects of the same type is usually faster than between objects of different types
- It's a good practice to represent the robot by a combination of basic primitives than using a full CAD model


Collision-detection
$\sim$ 100 triangles/robot


Visualization, from CAD
$\sim$ 10k triangles + textures/robot

Local planners

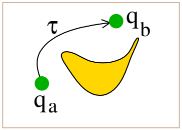
- Sampling-based planners rely on a “local planner”
- Given configurations $q_a \in \mathcal{C}_{\text{free}}$ and $q_b \in \mathcal{C}_{\text{free}}$, local planner attempts to find a path τ:

$$\tau : [0, 1] \rightarrow \mathcal{C}_{\text{free}}$$

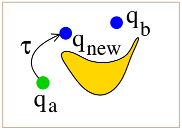
  such that $\tau(0) = q_a$ and $\tau(1) = q_b$, and τ must be collision free!
- Two-point boundary value problem (BVP)

Types of local planners (revision)

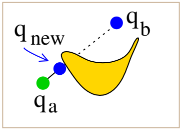
- Exact: analytic solution to BVP, e.g., Dubins or Reeds Shepp, straight-line (sometimes)
- Approximate: τ from q_a with q_{new} that is near-enough from q_b, e.g., straight-line
- Black-box models: physical simulation, e.g., for situations that cannot be solved analytically



Exact local planner



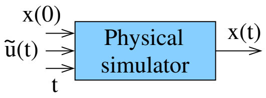
Approximate

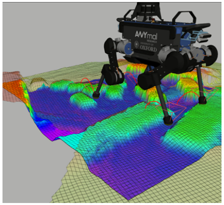


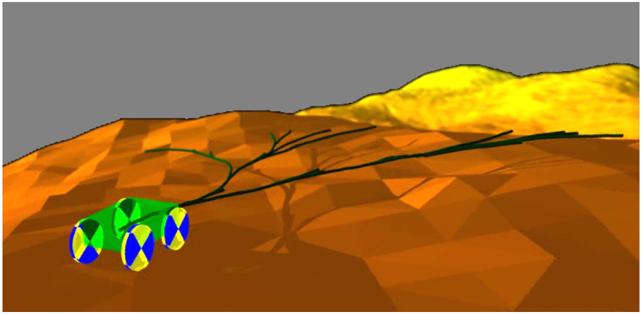
Straight-line

Local planner: System simulator

- Let’s assume a non-trivial scenario, e.g.,
  - mobile robot moving on a undulating terrain
  - or a legged robot walking on stones
- Analytic motion model is not easy to derive
- Instead, we can use a (physical) simulation
- Simulation is used as a “black-box”

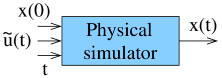


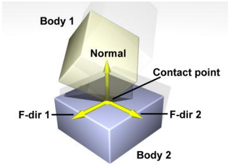


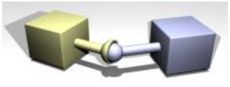


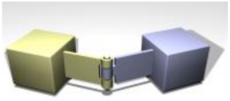
Physical simulation

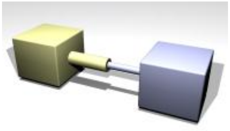
- Motion model of objects based on Newton physics
- Complex objects (robots) are composed of basic primitives
    - Spheres, Boxes, Cylinders
    - Analytic collision determination
- Each object has shape, mass and mass-density
- Objects are connected using static/movable joints
- Each join has limits/maximal moments, speed (+ internal states)
- Internal state s_i of object i: position, rotation, velocity, angular velocity

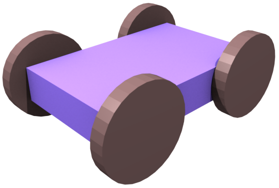








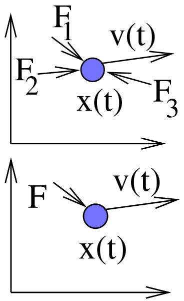


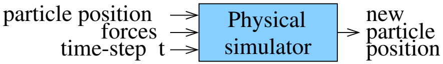


Physical simulation

Particle

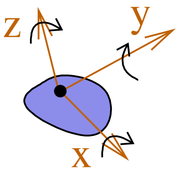
- Position $x(t)$, velocity $v(t)$, and mass m
- Various forces $\mathbf{F_i}$ are applied on the particle
- Particle movement is not constrained
- $\mathbf{F} = \sum \mathbf{F_i}$ is the total (net) force
- $\mathbf{F} = m\mathbf{a}(\mathbf{t})$, $\mathbf{a}(\mathbf{t}) = \dot{\mathbf{v}}(t)$, $\mathbf{v}(\mathbf{t}) = \dot{\mathbf{x}}(t)$
- Simulator computes $\mathbf{a}(t) \rightarrow \mathbf{v}(t) \rightarrow \mathbf{x}(t)$
- Integration over time-step ε (resolution of the simulation)
- Requires integration (Euler method, Midpoint, Runge-Kutta,. . .)
- Particle has no rotation





Physical simulation

Rigid body

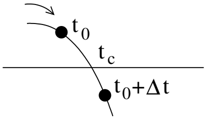


- Center of mass (CM) $\mathbf{x}(t)$, orientation $\mathbf{R}(t)$ (around CM)
- Translational velocity $\mathbf{v}(t)$ and angular velocity $\omega(t)$
- Net force $\mathbf{F}(t)$
- State vector is $\mathbf{y}(t) = (\mathbf{x}(t), \mathbf{R}(t), \mathbf{p}(t), \mathbf{L}(t))$
- $\mathbf{p}(t)$ is impulse, $\mathbf{L}(t)$ is angular momentum
- Constants: mass m, inertia tensor
- Unconstrained motion
- Leads to $\dot{\mathbf{y}}(t) = f(t, \mathbf{y}(t))$
- initial conditions: $y(t_0) = y_0$
- Solved by numerically

Simulating collisions

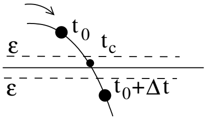
Colliding contact

- Particle is falling to a table
- Integration goes by step Δt: $t_0, t_0 + \Delta t, t_0 + 2\Delta t, \ldots$
- Integration is terminated if collision happens
- The time of collision t_c is estimated
- Change of velocities of colliding bodies is computed
- Simulation is started from t_c with new velocities
- This ensures instant change of velocities after the collision



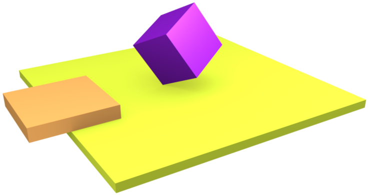
Detection of t_c assuming $t_0 < t_c < t_0 + \Delta t$

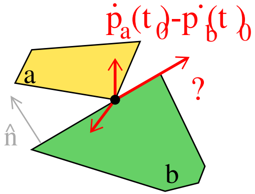
- Integration by intervals determined by the bisection method
- Alternatively, obtain collision depth from CD and accept t_c if the penetration depth is less than ε



Contacts

- Vertex/face
  - Vertex of one object is in contact with face of the other one
  - The Normal vector of the face determine the ‘normal of the contact’ $\hat{n}$
- Edge/Edge
  - Two edges *ea* and *eb* (each from different object) are in collision
  - $\hat{n} = ea \times eb$ (*ea* and *ev* are unit vectors)



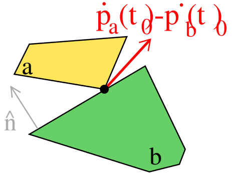


- Contacts $p_a(t_0)$ and $p_b(t_0)$, their velocity is $\dot{p}_a(t_0)$ and $\dot{p}_b(t_0)$
- $v_{rel} = \hat{n}(t_0) \cdot (\dot{p}_a(t_0) - \dot{p}_b(t_0))$
- Value of v_{rel} determines the type of collision

Contacts

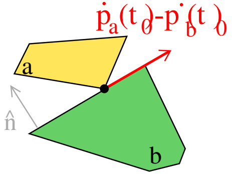
Separating

- $v_{rel} > 0$: bodies moving apart
- No reaction is needed



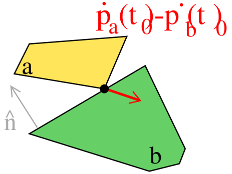
Contact/resting

- $v_{rel} = 0$
- No reaction is needed



Colliding

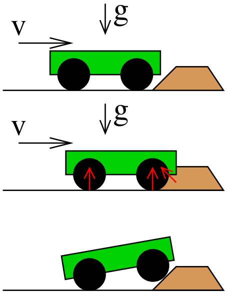
- $v_{rel} < 0$
- Compute the separating (penetration) vector, apply force to separate the objects
- Penetration vectors are not unique for non-convex objects
- The possible source of unstable simulation



Main loop of simulator

Particle

1. Create objects, create joints, . . .
2. User callback (read/set variables, display, . . .)
3. Apply forces
4. Update velocities and positions
5. Detect collisions
6. Solve constraints
7. Goto 2



Physical engines (sw. libraries)

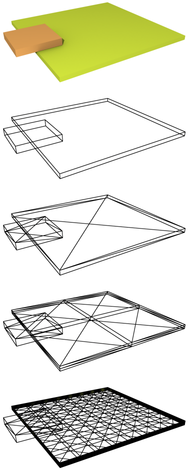
- Box2D, Chimpunk physics engine (2D)
- ODE, Bullet, Newton Game Physics (3D)

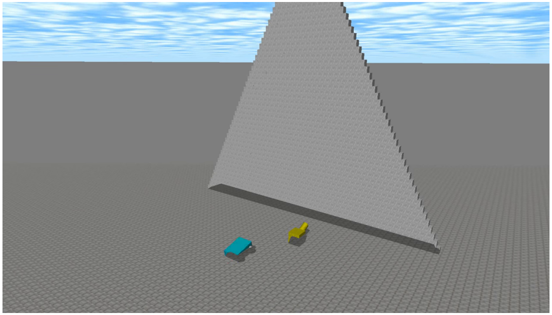
Robotic simulators (usually with GUI)

- They use physical engine inside, but offer more functionalities:
- Visualization, tools for interactive design of robots, import/export from URDF, sensors
- Gazebo, V-Rep (now CoppeliaSim), Webots, Player/Stage

Common issues of physical simulation

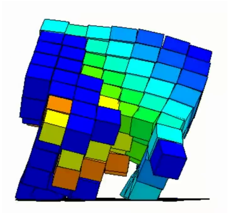
- If wrongly set up, it can “explode” or “freeze”
- Wrongly set up hinges, unrealistic masses, no gravity
- Too complex geometries → too complex (slow) collision detection
- Wrong friction parameters
- It’s better to prefer convex shapes (or composition of them) if possible

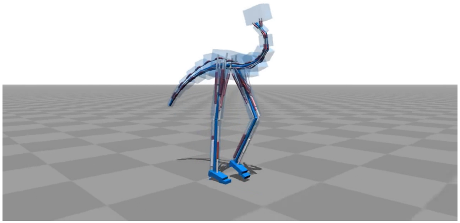




Physical simulation in other fields

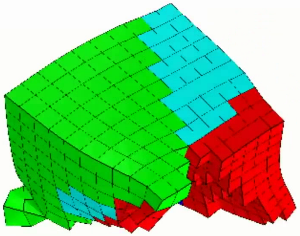
- 3D design/CAD simulation — design a machine and see how it works
- Virtual reality — e.g. for realistic object manipulation
- Computer games — realistic behavior of objects (without programming it)
- Evolving robots — evolutionary approaches to design robots or their parts, simulation serves as the fitness function evaluator





Physical simulation in other fields

In 2013, we saw simulated robots made of soft voxel cells evolve the ability to run.



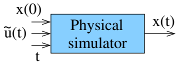
Cheney, N., MacCurdy, R., Clune, J., & Lipson, H. (2013). Unshackling evolution: evolving soft robots with multiple materials and a powerful generative encoding. In Proceeding of the fifteenth annual conference on genetic and evolutionary computation (pp. 167-174). ACM.

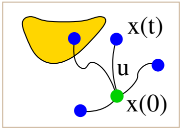
Physical simulation as local planner

- Most of physical simulators (ODE, Bullet and their derivatives) assume time-linear simulation
- In motion planning, we need a non-linear simulation
- We need to "restart" simulation for each tree expansion

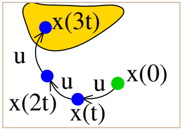
RRT with system simulator

- Each node contains: $x = (s_i), i = 1, \ldots n$ (simulator state)
- Tree expansion from node $x_{\text{near}} = x(0)$ using input u
  - Set simulator to state x_{near} (restart)
  - Apply control inputs u (usually joint moments)
  - Run simulation for time Δt
  - Read simulator state x
  - Add node x to the tree
- Usually several control inputs $u \in \mathcal{U}$ is tested

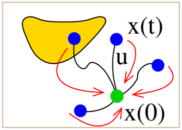




We need



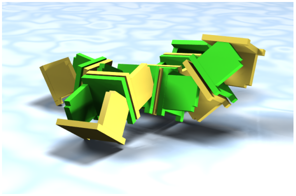
No restarts

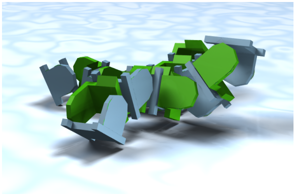


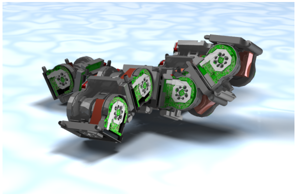
With restarts

Physical simulation: tricks

- Try to minimize number of objects/joints
- Avoid using triangle mesh for collision-detection
  - Physical simulation needs collision determination (penetration vector)
  - CD may be unstable on (non-convex) meshes, simulation can "explode"
- If possible, approximate robots by boxes/spheres/cylinders $\rightarrow$ fast and stable collision detection
- Use separate models for physics and visualization


Mass
10 boxes/robot


Collision-detection
$\sim$ 100 triangles/robot


Visualization
$\sim$ 10k triangles + textures/robot