Path Planning

15-494 Cognitive Robotics
David S. Touretzky &
Ethan Tira-Thompson

Carnegie Mellon
Spring 2013

Outline

- Path planning as state space search
- RRTs: Rapidly-exploring Random Trees
- The RRT-Connect algorithm
- Collision detection
- Smoothing
- Path planning with constraints
- Path planning in Tekkotsu

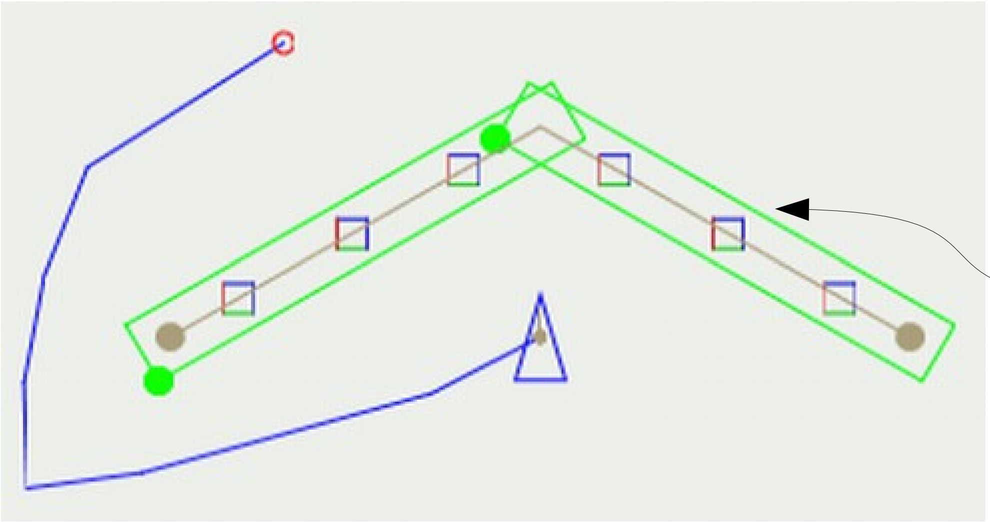
Path Planning in Robotics

1. Navigation path planning
    - How to get from the robot's current location to a goal.
    - Avoid obstacles.
    - Provide for localization.

2. Manipulation path planning
    - Move an arm to grasp and manipulate an object.
    - Avoid obstacles.
    - Obey constraints (e.g., don't spill the coffee).

Navigation Planning

- 2D state space: (x,y) coordinates of the robot
  - Treat the robot as a point or a circle.



Obstacle
inflation

- 3D state space: (x,y,θ) pose of the robot
  - Heading matters when the robot is asymmetric
  - Heading matters when the robot's motion is constrained

Cspace Transform

- The area around an obstacle that would cause a collision with the robot.

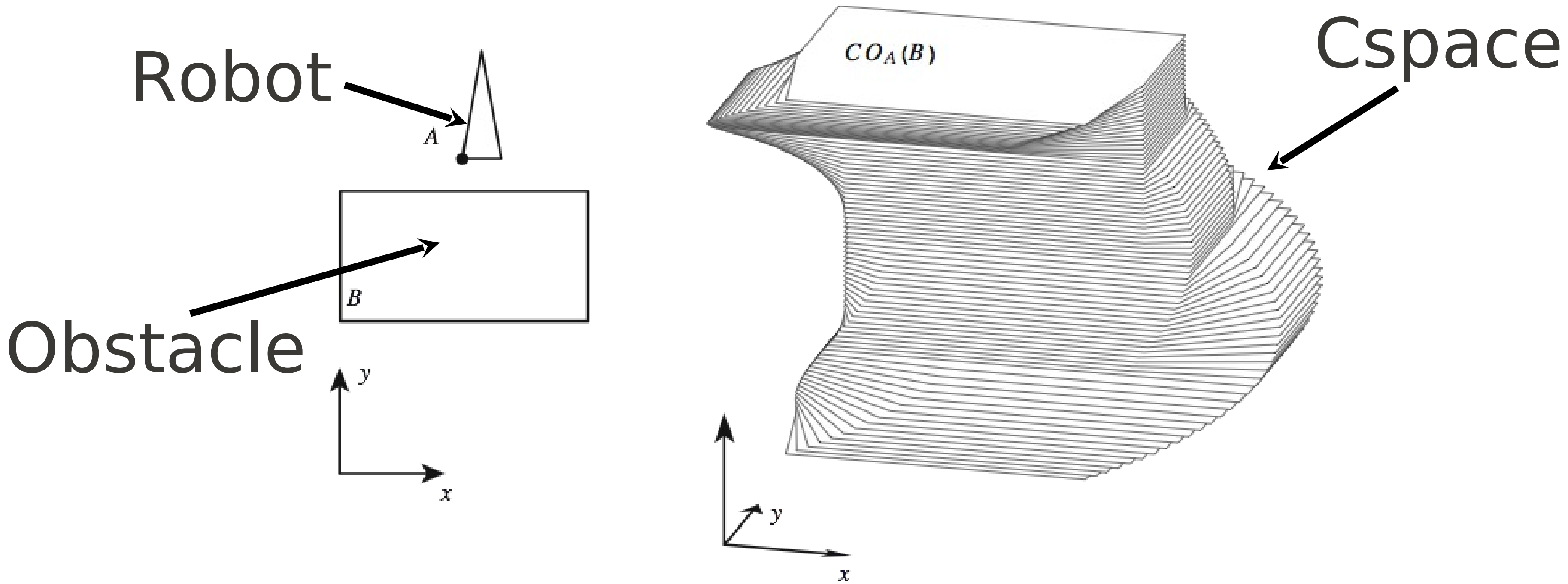


Figure 4.4 - Mason, Mechanics Of Robotic Manipulation

Arm Path Planning

- Cspace transform blocks out regions of joint space

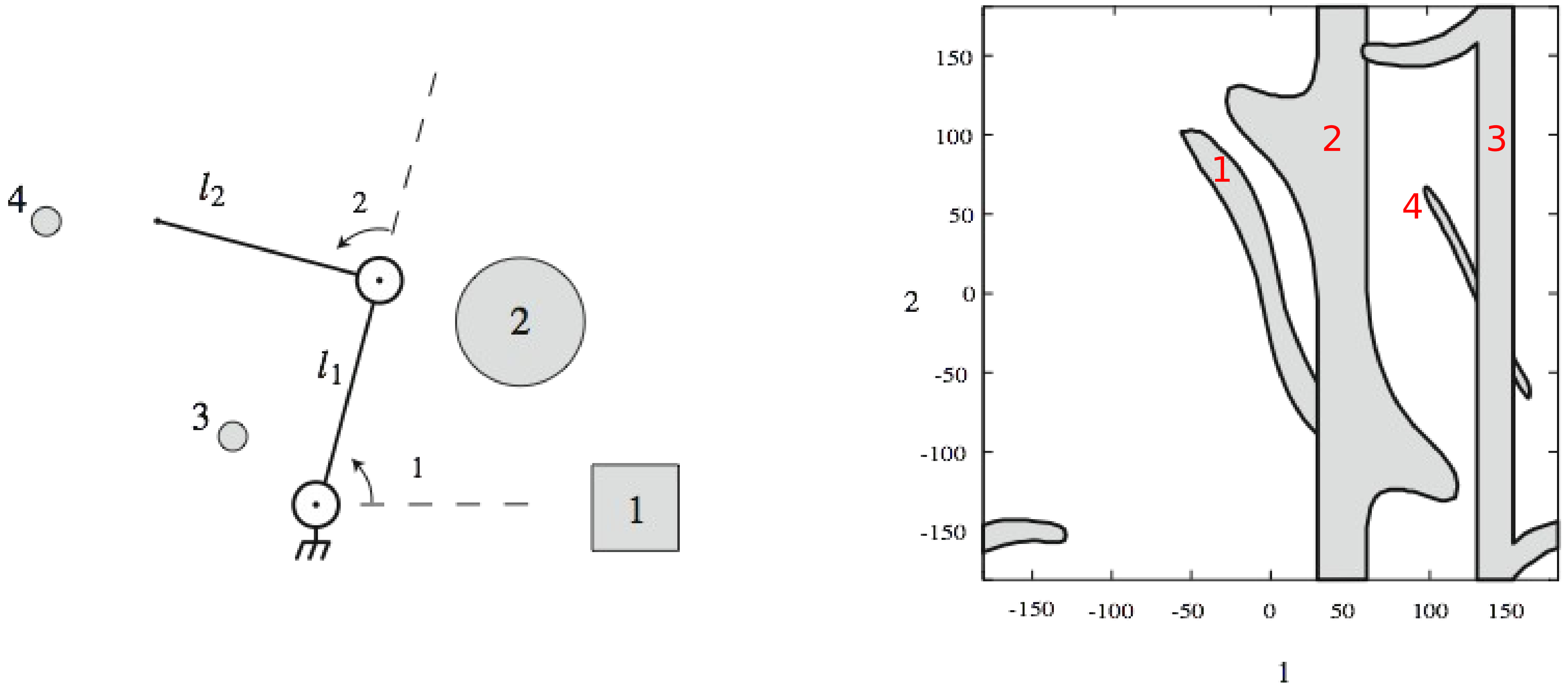


Figure 4.5 - Mason, Mechanics Of Robotic Manipulation

State Space Search

The path planning problem:

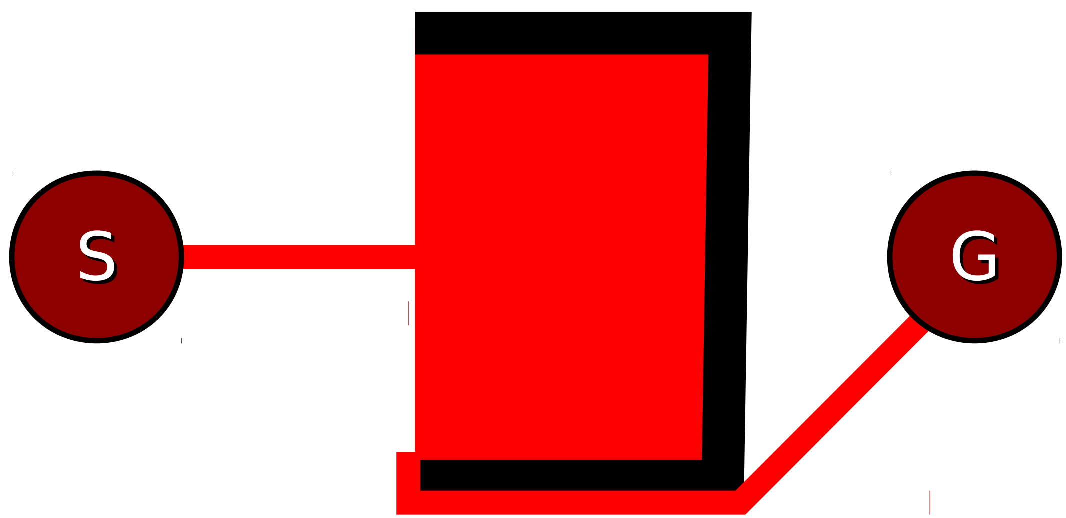
Given an n-dimensional state space and

- a start state $S=[s_1,s_2,\ldots,s_n]$
- a goal state $G=[g_1,g_2,\ldots,g_n]$
- an admissibility predicate P (collision test + constraints)

find a path from S to G such that every state on the path satisfies P.

Best First Search

- Can get trapped in a cul de sac for a long time.



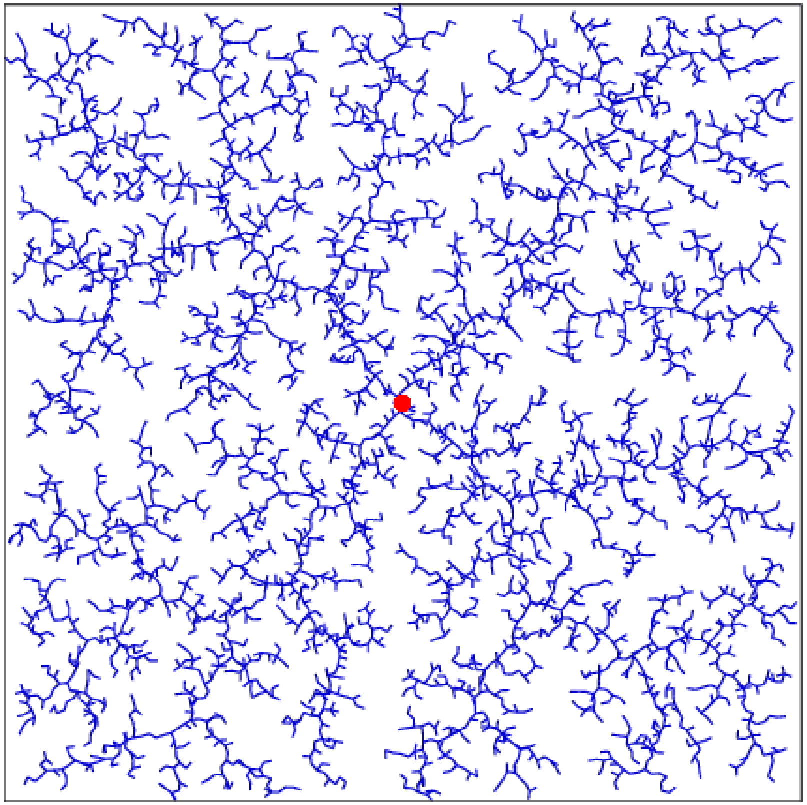
- Random search might be faster.

Rapidly-exploring Random Trees

- Described in LaValle (1998), Kuffner & LaValle (2000)
- Create a tree with start state S as the root.
- Repeat up to K times:
  - Pick a point **q** in configuration space:
    - Sometimes **q** should be a random point
    - Sometimes **q** should be the goal state G
  - Find **n**, the closest tree node to **q**
  - Add a new node **n'** some distance Δ toward **q**; make it a child of **n**
  - If **n'** is close enough to the goal state G, return.

RRT Algorithm

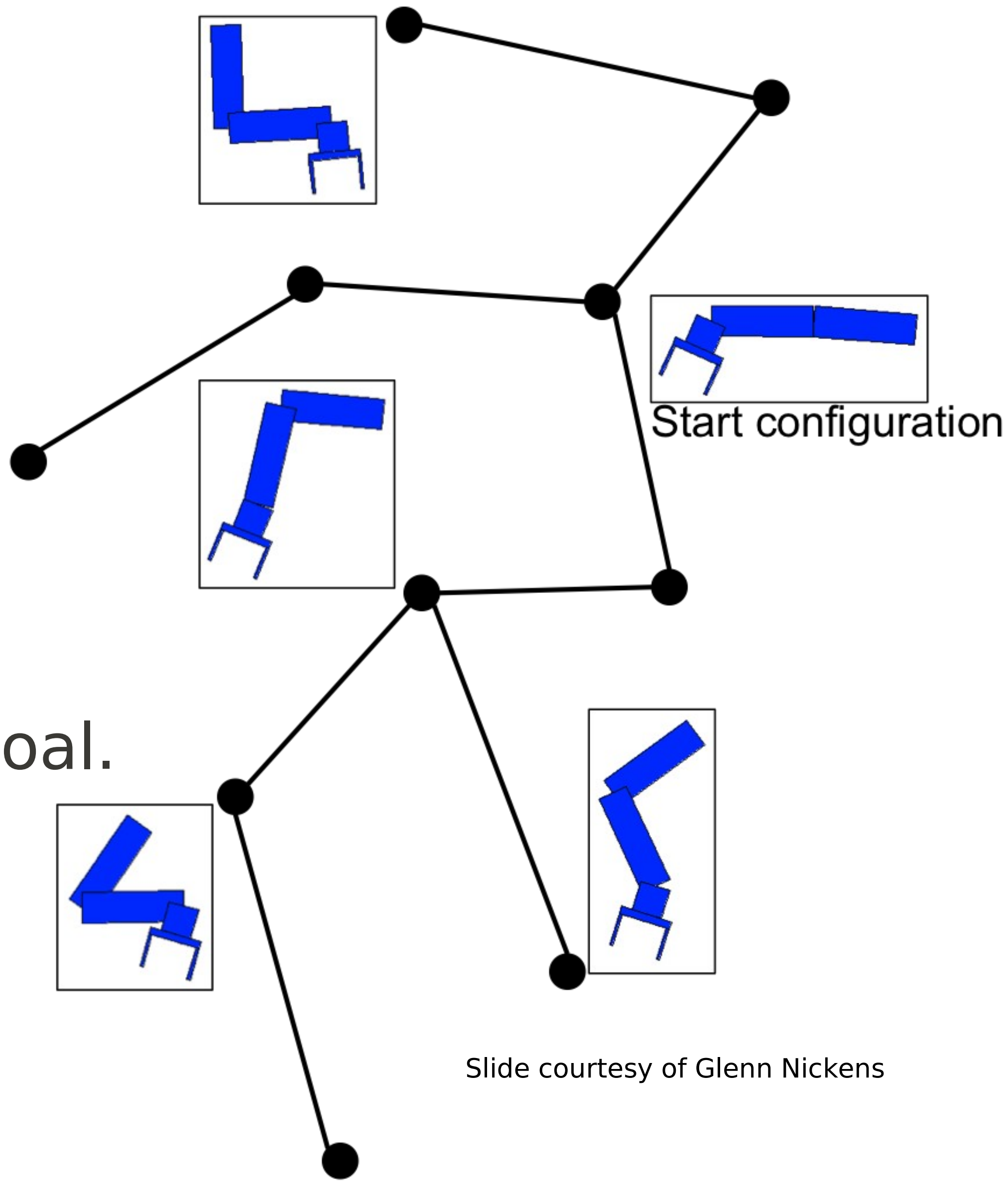
- Rapidly samples the state space.
- Cannot get trapped in local minima.
- Works well in high-dimensional spaces.
- Does not generate smooth paths.
- Can't tell when no solution exists; only quits when it exceeds the iteration limit K.



http://msl.cs.uiuc.edu/rrt/treemovie.gif

RRTs for Arm Path Planning

- Each node encodes an arm configuration in joint space.
- Only add nodes that don't cause collisions (with self or obstacles).
- Alternately (i) extend the tree in random directions and (ii) move toward the goal.



Slide courtesy of Glenn Nickens

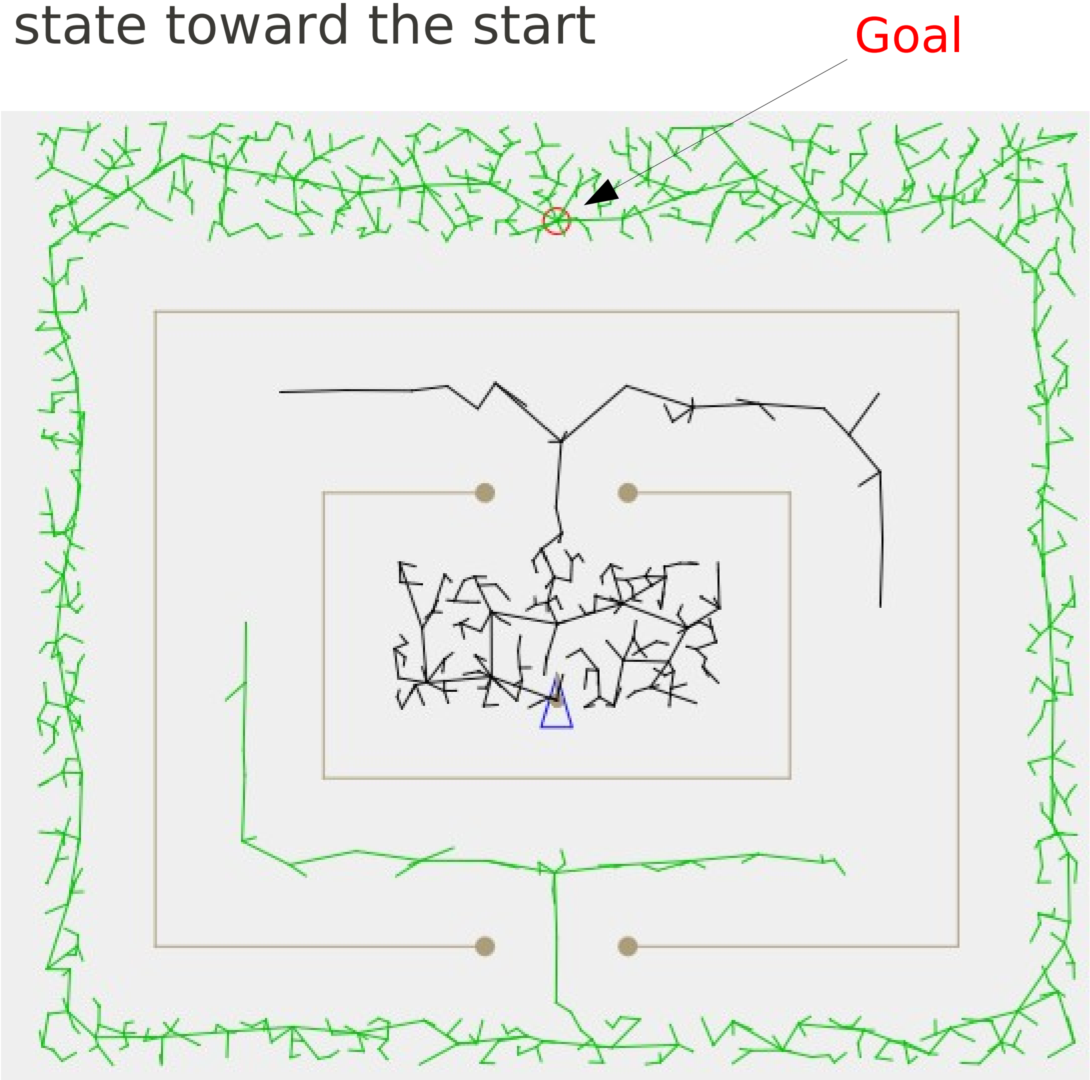
Implementation Notes

- Finding **n**, the nearest node in the tree to **q**, is the most expensive part of the algorithm.
  - Use K-D trees to efficiently find **n**?
  - In practice, K-D trees are slower unless you have a huge number of nodes (several thousand).

- Why only go a distance Δ toward the goal state G? Why not go as far as we can, in steps of Δ?
  - With no obstacles, this reaches the goal very quickly, but random search will get there nearly as quickly as we keep extending the nearest node to the goal.
  - But when obstacles are present, this can waste time filling out branches that will ultimately fail.
  - Generating lots of extra nodes bloats the tree, which slows down the algorithm.

RRT-Connect Algorithm

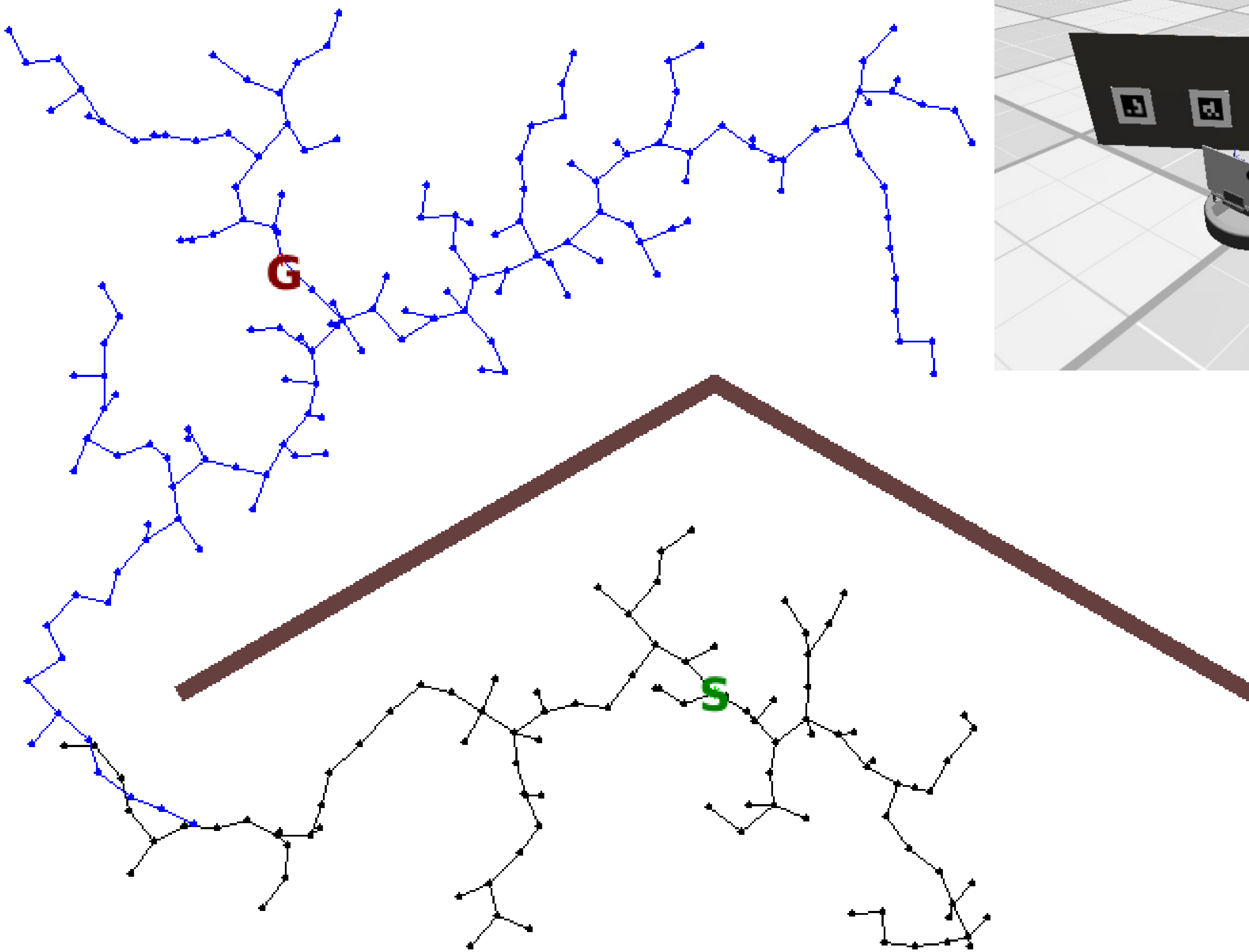
- Variant of RRT that grows two trees:
  - one from the start state toward the goal
  - one from the goal state toward the start

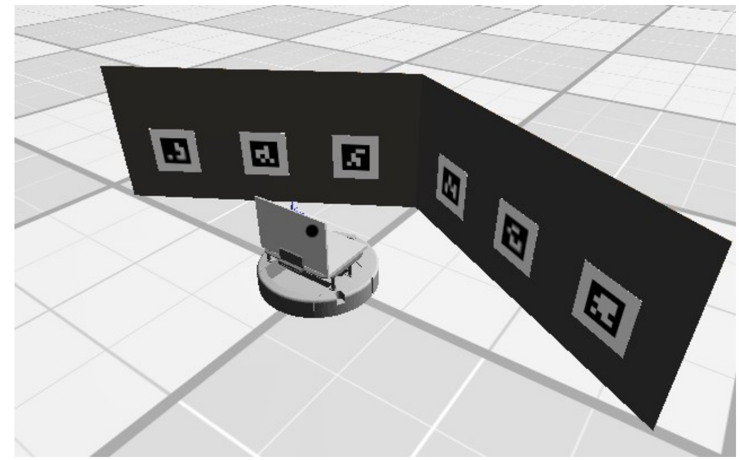


- When the two trees connect, a solution has been found.

- Not guaranteed to be better than RRT, but often helps.

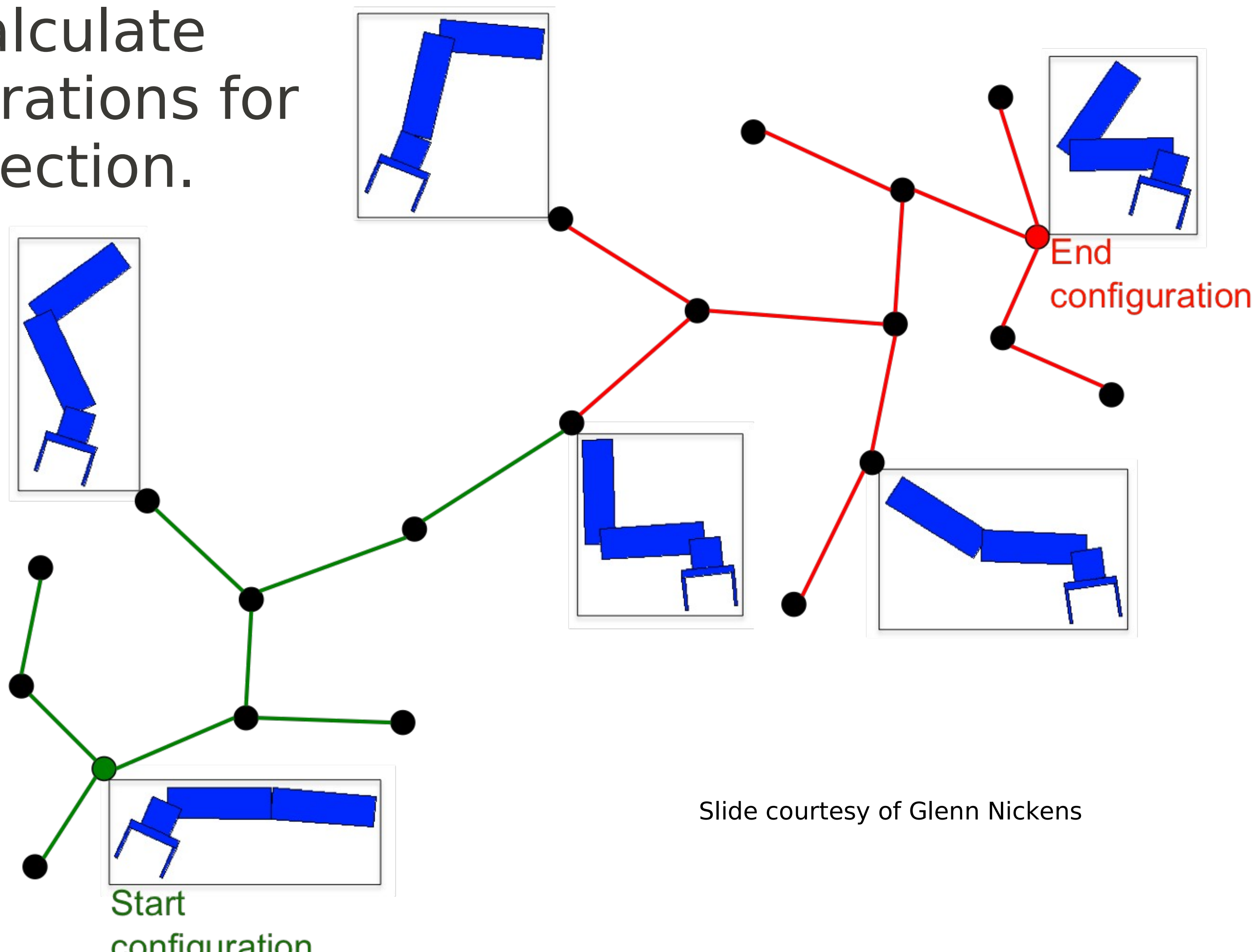
RRTs in the VeeTags World





RRT-Connect For Arms

- Use IK to calculate the goal configuration.
- Use FK to calculate arm configurations for collision detection.



Slide courtesy of Glenn Nickens

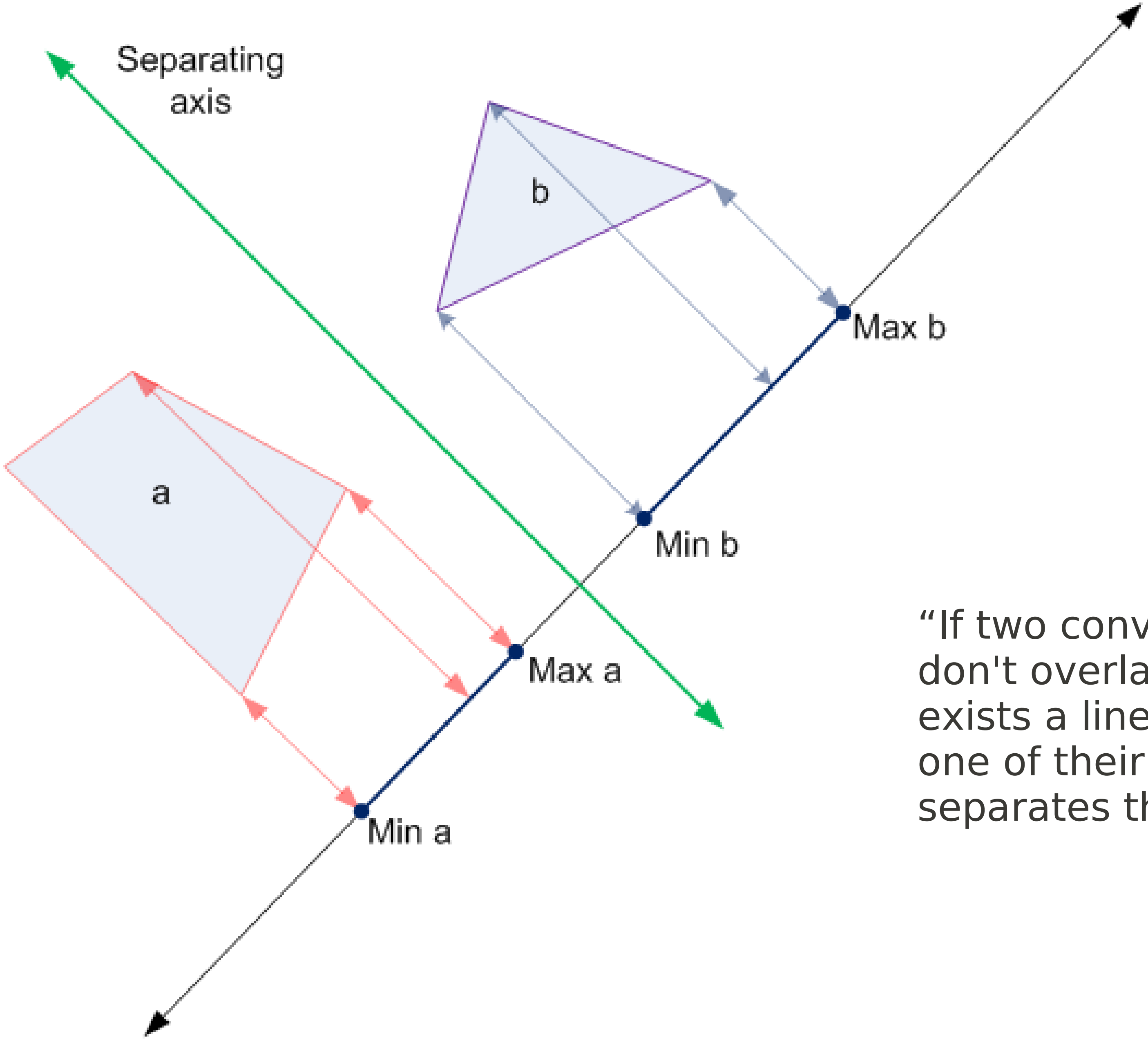
Collision Detection

- Represent the robot and the obstacles as **convex polygons**.

- In 2D, use the Separating Axis Theorem to check for collisions.
  - Easy to code
  - Fast to compute

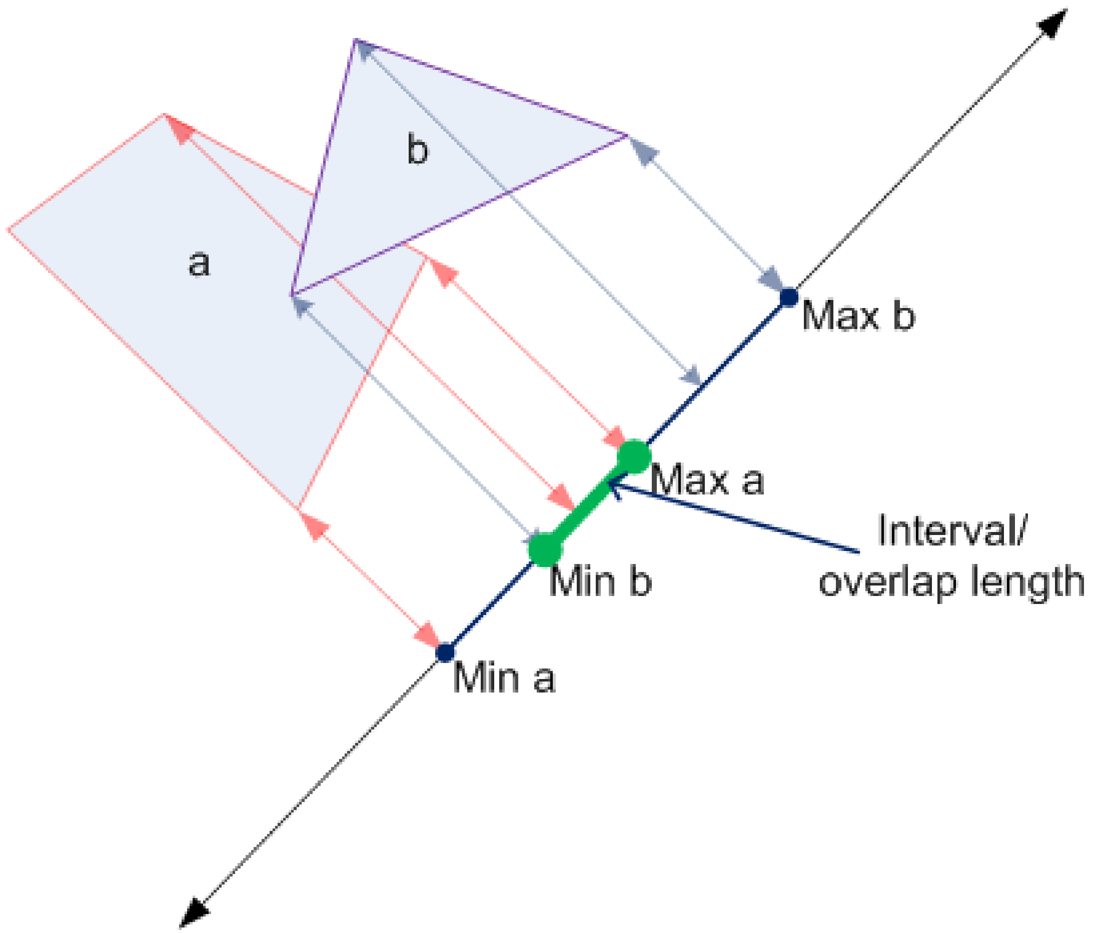
- In 3D, things get more complex.
  - Tekkotsu uses the GJK (Gilbert-Johnson-Keerthi) algorithm, used in many physics engines for video games.

Separating Axis Theorem



“If two convex polygons don't overlap, then there exists a line, parallel to one of their edges that separates them.”

Separating Axis Theorem



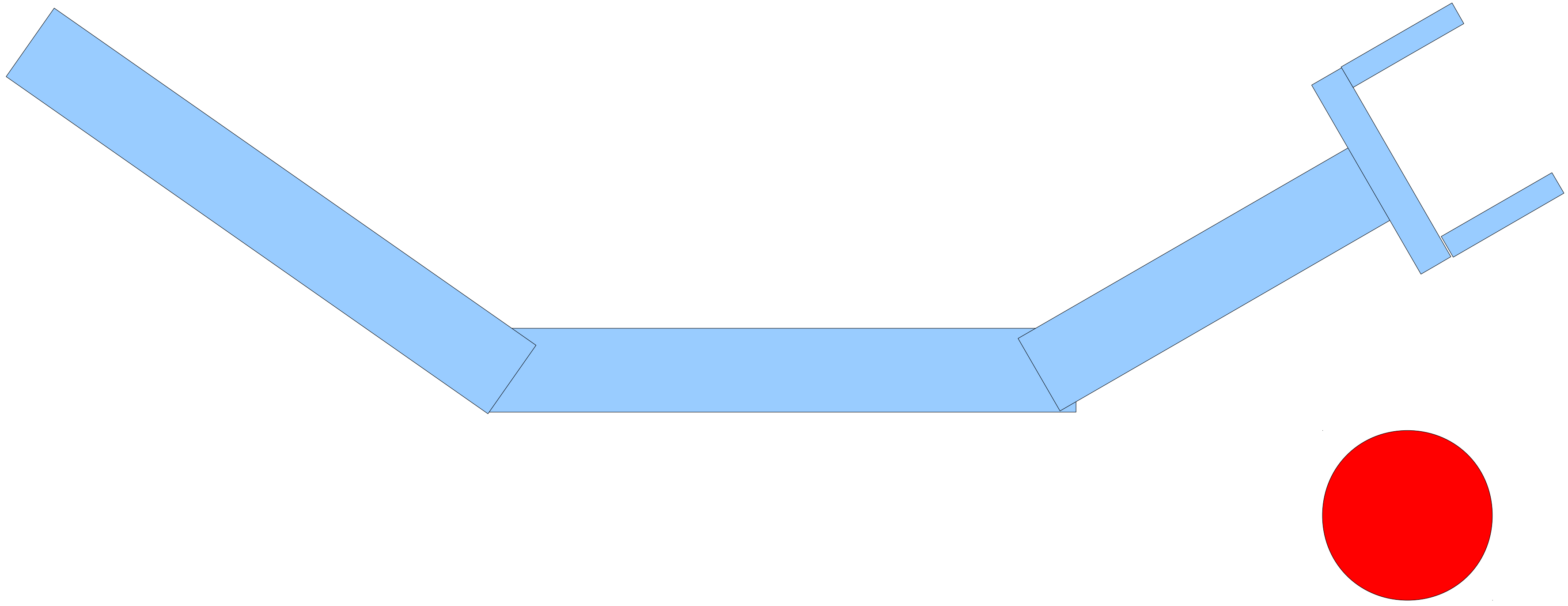
Algorithm to Apply the SAT

- For every edge of polygon A and of polygon B:
  - Project all the vertices onto the line normal to that edge.
  - Calculate the min and max coordinates for each polygon
  - If minA < minB and maxA > minB OR
    if minB < minA and maxB > minA
    then the polygons collide.

- If you find any edge projection in which the ranges don't overlap, the polygons do not collide.

Arm Collision Detection

- Represent each link as a separate polygon.
- Check for:
  - Self-collisions other than link *n* with link *n+1*
  - Collisions of a link with an obstacle



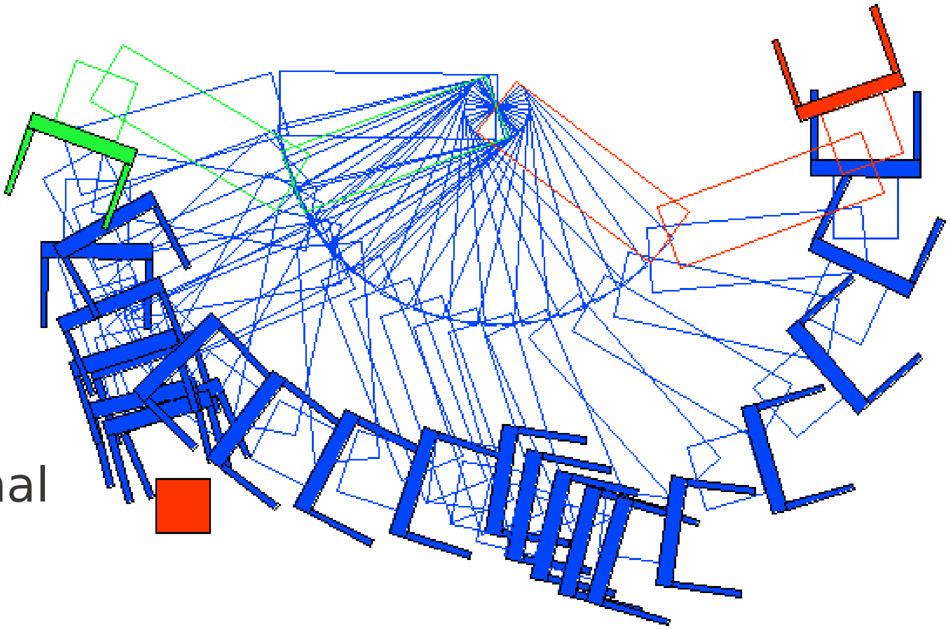
Path Smoothing

- The random component of RRT-Connect search often results in a jerky and meandering solution.
- Solution: apply a path smoothing algorithm.
- Repeat N times:
  - Pick two points on the path at random
  - See if we can linearly interpolate between those points without collisions.
  - If so, then snip out that segment of the path.

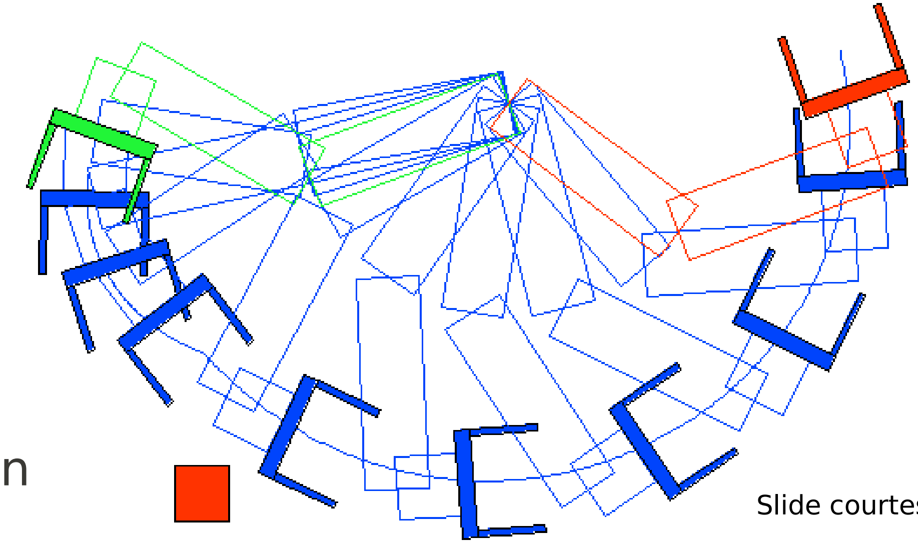
Smoothing An Arm Trajectory

- Start state
- Intermed. states
- End state

Original



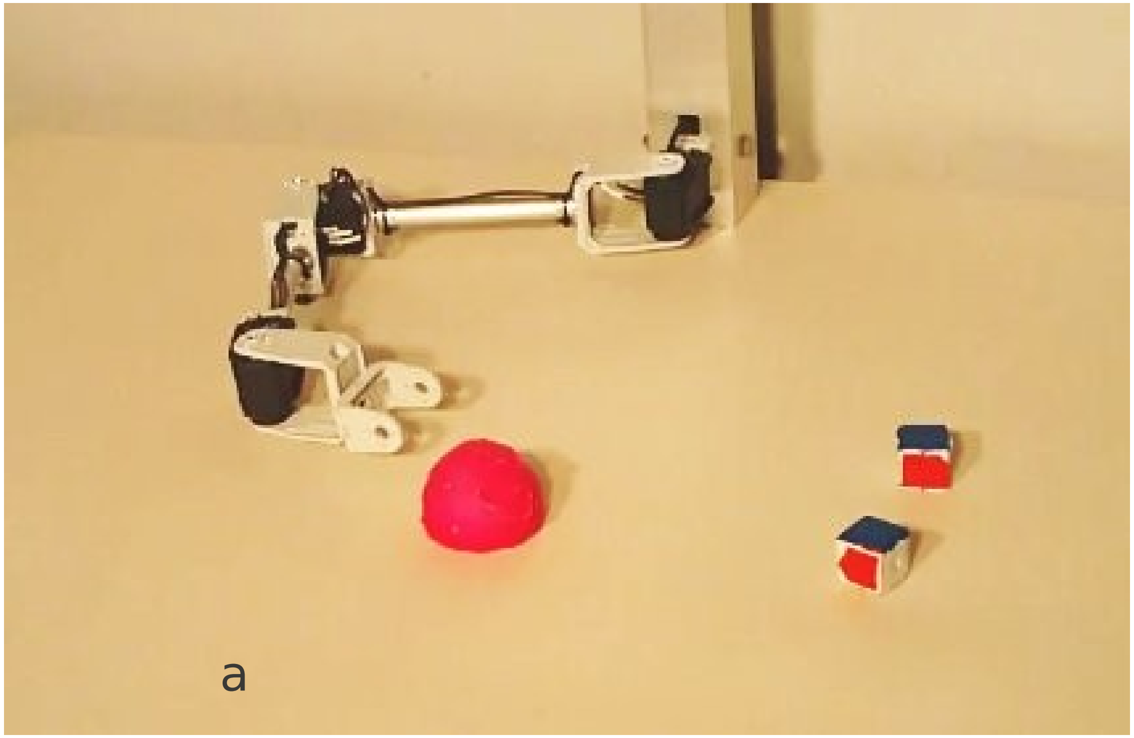
Smoothed version



Slide courtesy of Glenn Nickens

Path Planning With Constraints

- With no closeable fingers, arm motion is constrained to be within about 60° of finger direction or we'll lose the object.



(video)

http://www.youtube.com/watch?v=9oDQ754YVoc

Implementing Constraints

- Each time we generate a new state **n'**:
  - Check to see if **n'** obeys the constraint
  - For finger motion constraint, check if the direction of motion from parent state **n** to new state **n'** is within 60° of the finger direction.
- What if **n'** doesn't obey the constraint?
  - Reject it and generate a new random **q**.
  - Or try to “fix” it by perturbing its value slightly so as to satisfy the constraint.

RRTs in Tekkotsu

- Tekkotsu/Planners/RRT/GenericRRT.h
- Works for any state space

- class RRTNodeBase
  - Subclass this to create a NodeValue_t to describe **q**
  - Define a CollisionChecker class

- class GenericRRT<typename NODE, size_t N>
  - Instantiate this template to create an RRT planner
  - NODE must be a subclass of RRTNodeBase
  - Define an AdmissibilityPredicate class
  - Define the extend(...) method to extend the tree

Planners in Tekkotsu

- Navigation/ShapeSpacePlannerXY
  - 2D navigation planner
- Navigation/ShapeSpacePlannerXYTheta
  - 2D + heading navigation planner
- Manipulation/ShapeSpacePlanner2DR
  - 2D planner for N-joint planar arm with revolute joints
- Manipulation/ShapeSpacePlanner3DR
  - 3D planner for N-joint planar arm with revolute joints

The Grasper

- Does arm path planning
  - Currently only for planar arms
  - 3D arm path planning coming soon
- Does manipulation planning
  - How to grasp an object
  - How to move an object without losing it
  - How to release an object
- Many other manipulation operations are possible.
- Use a GrasperNode to submit a GrasperRequest.