Approximate path planning

Computational Geometry
csci3250
Laura Toma
Bowdoin College

Path planning

- Combinatorial <—— last time
- Approximate <—— next

Combinatorial path planning

- **Idea: Compute free C-space combinatorially (= exact)**
- **Approach**
  - (robot, obstacles) => (point robot, C-obstacles)
  - Compute roadmap of free C-space
    - any path: trapezoidal decomposition or triangulation
    - shortest path: visibility graph
- **Comments**
  - Complete
  - Works beautifully in 2D and for some cases in 3D
    - Worst-case bound for combinatorial complexity of C-objects in 3D is high
  - Unfeasible/intractable for high #DOF
    - A complete planner in 3D runs in $O(2^{n\wedge\#DOF})$

Approximate path planning

- **Since you can’t compute C-free, approximate it**

- Approaches
  - Graph search strategies
    - A*, weighted A*, D*, ARA*,…
  - Sampling-based + roadmaps
    - PRM, RRT, …
  - Potential field
  - Hybrid

Planning as a graph-search problem

1. Construct the graph representing the environment

2. Search the graph (hopefully for a close-to-optimal) path

The two steps are often interleaved.

Planning as a graph-search problem

Graphs can be grouped into two classes

- **Skeletonization/roadmaps**
    - visibility graphs
    - trapezoidal decomposition
    - (probabilistic) roadmaps
- **Cell decomposition**
    - grids
    - multi-resolution grids

Grid-based graphs

- Sample C-space with uniform grid/lattice
  - refined: quadtree/octree
  - This essentially “pixelizes" the space (pixels/voxels in C-free)
- Graph is usually implicit
  - given by lattice topology: move +/-1 in each direction, possibly diagonals as well
  - successors(state s)
- Search the graph for a path from start to end
  - Dijkstra/A* + variants
- Graph can be pre-computed (occupancy grid), or computed incrementally
  - one-time path planning vs many times
  - static vs dynamic environment

Dijkstra's SSSP algorithm

- best-first search: priority(v) = d[v] = cost of getting from s to v
- initialize: d[v] = inf for all v, d[s] = 0
- repeat: greedily select the vertex with smallest priority, and relax its edges

Dijkstra(vertex s)

- initialize
    - d[v] = infinity for all v, d[s] = 0
- for all v: PQ.insert(<v, d[v]>)
- while PQ not empty
    - u = PQ.deleteMin()
    - //claim: d[u] is the SP(s,u)
    - for each edge (u,v):
        - if v not done, and if d[v] > d[u] + edge(u,v):
            - d[v] = d[u] + edge(u,v)
            - PQ.decreasePriority(v, d[v])

no need to check if v is done,
because once v is done,
no subsequent relaxation can improve its d[]

usually not implemented

Dijkstra(vertex s)

```
• initialize
    • d[v] = infinity for all v, d[s] = 0
• PQ.insert(<s, d[s]>)
• while PQ not empty
    • u = PQ.deleteMin()
    • for each edge (u,v):
        • if isFree(v) and d[v] > d[u] + edge(u,v):
            • d[v] = d[u] + edge(u,v)
            • PQ.insert(<v, d[v]>)
```

insert only the start

insert it
(even if it’s already there)

isFree(v): is v in C-free

What to do with a partially blocked cell?

Grid-based graphs

- Dijkstra's algorithm :   if only one path is needed (not SSSP), a heuristic can be used to guide the search towards the goal

- A*
  - best-first search
  - **priority f(v) = g(v) + h(v)**
    - g(v): cost of getting from start to v
    - h(v): estimate of the cost from v to goal
  - Theorem: If h(v) is “admissible” (h(v) < trueCost(v—>goal)) then A* will return an optimal solution.
  - Dijkstra is (A* with h(v) = 0)
  - In general it may be hard to estimate h(v)
    - path planning: h(v) = EuclidianDistance(v, goal)

Grid-based graphs

- A* explores fewer vertices to get to the goal, compared to Dijkstra
  - The closer h(v) is to the trueCost(v), the more efficient
- Example
  - https://www.youtube.com/watch?v=DINCL5cd_w0

- Many A* variants
  - weighted A*
    - c x h() ==> solution is no worse than (1+c) x optimal
  - real-time replanning
    - if the underlying graph changes, it usually affects a small part of the graph ==> don't run search from scratch
    - D*: efficiently recompute SP every time the underlying graph changes
  - anytime A*
    - use weighted A* to find a first solution ; then use A* with first solution as upper bound to prune the search

Grid-based graphs

- Comments
  - Not complete, but **resolution complete**
    - find a solution, if one exists, with probability —> 1 as the resolution of the grid increases
  - The paths may be longer than true shortest path in C-space
  - can interleave the construction with the search (ie construct only what is necessary)
  - +
    - very simple to understand/implement
    - works in any dimension
  - -
    - size depends on size of the environment
    - size can be too large => slow

Sampling-based planning

- Combinatorial: hard to construct C-obstacles exactly when D is high
- Grid-based: space is too large when D is high
- Sampling-based planning
  - generates a sparse (sample-based) representation of free C-space
  - isFree(p): would my robot , if placed in this configuration, intersect any obstacle?
  - points to sample are generated probabilistically
  - aims to provide probabilistic completeness
    - find a solution, if one exists, as the number of samples approaches infinity
  - well-suited for high D planning

Sampling

- You are not given the representation of C-free: Imagine being blindfolded in a maze

- Sampling: you walk around hitting your head on the walls

- Left long enough, after hitting many walls, if you remember everything, you have a pretty good representation of the maze

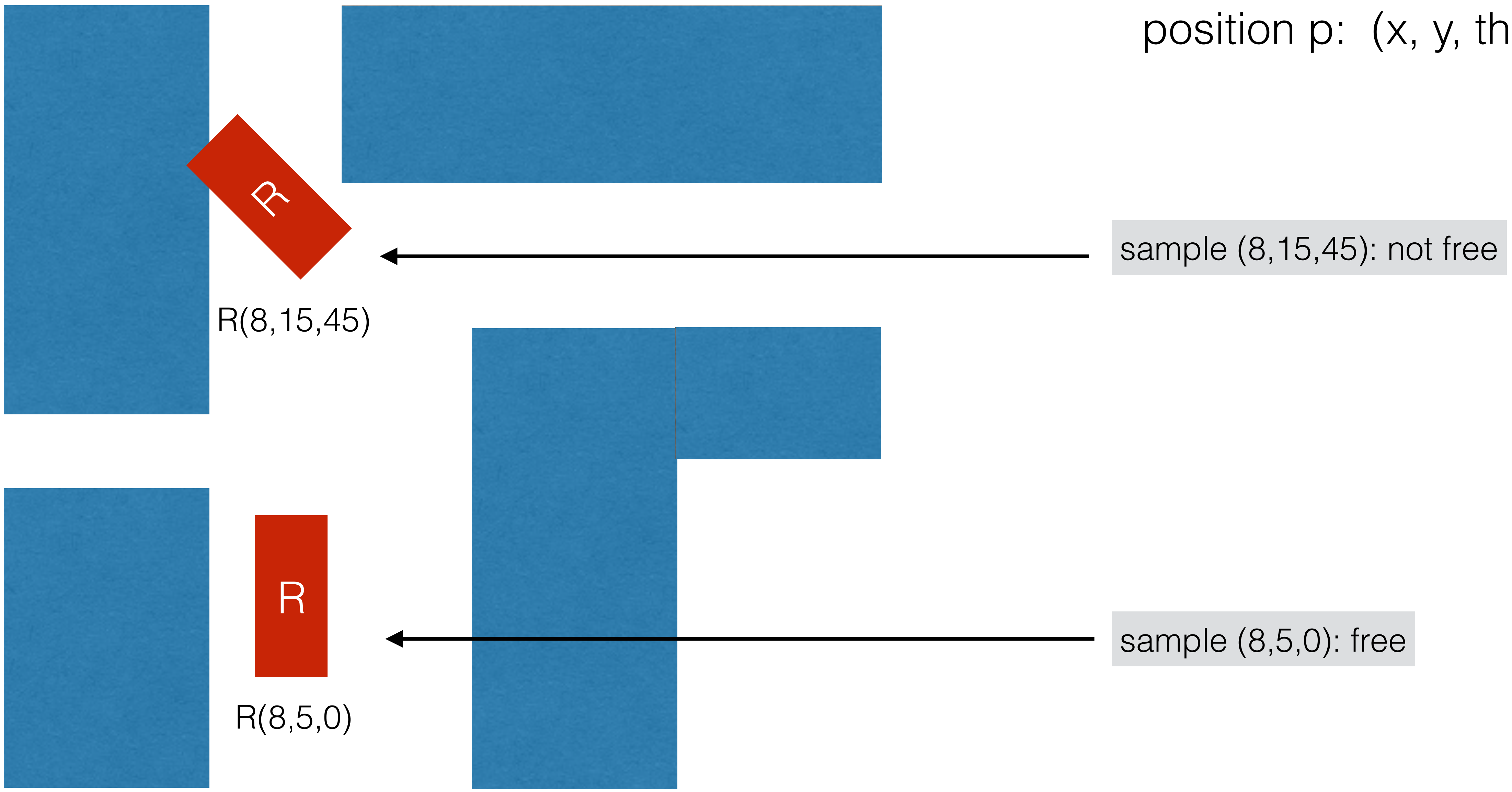
- However the space is huge
  - e.g. DOF= 6: 1000 x 1000 x 1000 x 360 x 360 x 360

- So you need to be smart about how you chose the points to sample

2D: robot can translate and rotate

C-space: 3D

position p: (x, y, theta)



How would you write: isFree((x,y,theta)) ?

GENERIC sampling-based planning

- Roadmap
  - Instead of computing C-free explicitly, sample it and compute a roadmap that captures its connectivity to the best of our (limited) knowledge
- Roadmap construction phase
  - Start with a sampling of points in C-free and try to connect them
  - Two points are connected by an edge if a simple quick planner can find a path between them
  - This will create a set of connected components
- Roadmap query phase
  - Use roadmap to find path between any two points

GENERIC sampling-based planning

- Generic-Sampling-based-roadmap:
  - V = p_{start} + sample_points(C, n); E = { }
  - for each point x in V:
    - for each neighbor y in neighbors(x, V):

      //try to connect x and y

      - if collisionFree(segment xy): E = E + xy
  - return (V, E)

- Algorithms differ in
  - sample_points(C, n) : how they select the initial random samples from C
    - return a set of n points arranged in a regular grid in C
    - return random n points
  - neighbors(x, V) : how they select the neighbors
    - return the k nearest neighbors of x in V
    - return the set of points lying in a ball centered at x of radius r
  - Often used: samples arranged in a 2-dimensional grid, with nearest 4 neighbors (d, 2^d)

Probabilistic Roadmaps (Kavraki, Svetska, Latombe, Overmars et al , 1996)

- Start with a *random* sampling of points in C-free
- Roadmap stored as set of *trees* for space efficiency
    - trees encode connectivity, cycles don’t change it. Additional edges are useful for shortest paths, but not for completeness
- Augment roadmap by selecting additional sample points in areas that are estimated to be “difficult”

```
N ← ∅
E ← ∅
loop
    c ← a randomly chosen free
     configuration
    N_c ← a set of candidate neighbors
      of c chosen from N
    N ← N ∪ {c}
    for all n ∈ N_c, in order of
     increasing D(c,n) do
      if ¬same_connected_component(c,n)
       ∧Δ(c,n) then
               E ← E ∪ {(c,n)}
               update R's connected
                components
```

- Components
    - sampling C-free: random sampling
    - selecting the neighbors: within a ball of radius r
    - the local planner delta(c,n): is segment cn collision free?
    - the heuristical measure of difficulty of a node

Probabilistic Roadmaps (Kavraki, Svetska, Latombe, Overmars et al , 1996)

Comments

- Roadmap adjusts to the density of free space and is more connected around the obstacles
- Size of roadmap can be adjusted as needed
- More time spent in the “learning” phase ==> better roadmap
- Shown to be **probabilistically complete**
    - probability that the graph contains a valid solution —> 1 as number of samples increases

```
N ← ∅
E ← ∅
loop
    c ← a randomly chosen free
     configuration
    N_c ← a set of candidate neighbors
      of c chosen from N
    N ← N ∪ {c}
    for all n ∈ N_c, in order of
     increasing D(c,n) do
      if ¬same_connected_component(c,n)
       ∧Δ(c,n) then
             E ← E ∪ {(c,n)}
             update R's connected
              components
```

Probabilistic Roadmaps

- One of the leading motion planning technique
- Efficient, easy to implement, applicable to many types of scenes
- Embraced by many groups, many variants of PRM's, used in many type of scenes.
  - PRM*
  - FMT* (fast marching tree)
  - …
- Not completely clear which technique better in which circumstances

-

Incremental search planners

- Graph search planners over a fixed lattice:
    - may fail to find a path or find one that's too long
- PRM:
    - suitable for multiple-query planners
- Incremental search planners:
    - designed for single-query path planning
    - incrementally build increasingly finer discretization of the configuration space, while trying to determine if a path exists at each step
    - probabilistic complete, but time may be unbounded

Incremental search planners

- RRT (LaValle, 1998)
- Idea: Incrementally grow a tree rooted at “start” outwards to explore reachable configuration space

```
BUILD_RRT(q_init)
  T.init(q_init);
  for k = 1 to K do
      q_rand ← RANDOM_CONFIG();
      EXTEND(T, q_rand);
  Return T
```

```
EXTEND(T, q)
  q_near ← NEAREST_NEIGHBOR(q, T);
  if NEW_CONFIG(q, q_near, q_new) then
      T.add_vertex(q_new);
      T.add_edge(q_near, q_new);
      if q_new = q then
          Return Reached;
      else
          Return Advanced;
  Return Trapped;
```

Figure 2: The basic RRT construction algorithm.

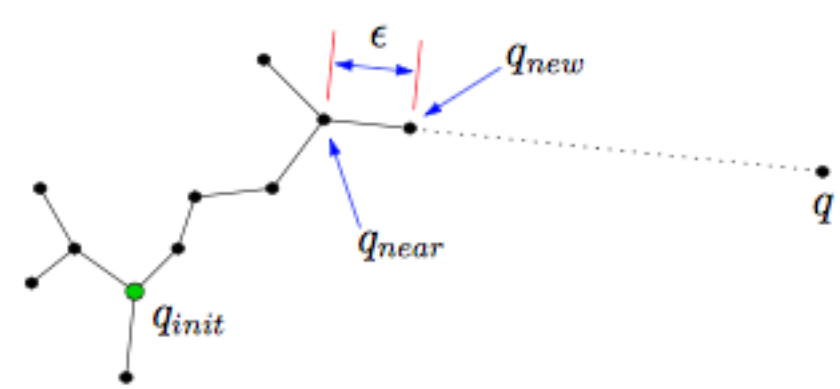


Figure 3: The EXTEND operation.

- https://personalrobotics.ri.cmu.edu/files/courses/papers/Kuffner00-rrtconnect.pdf

Self-driving cars

- Both graph search and incremental tree-based
- DARPA urban challenge:
  - **CMU**: lattice graph in 4D (x,y, orientation, velocity); graph search with D*
  - **Stanford**: incremental sparse tree of possible maneuvers, hybrid A*
  - **Virginia Tech**: graph discretization of possible maneuvers, searchwith A*
  - **MIT**: variant of RRT with biased sampling

Good read: *A Survey of Motion Planning and Control Techniques for Self-driving Urban Vehicles, by Brian Paden, Michal Cáp, Sze Zheng Yong, Dmitry Yershov, and Emilio Frazzoli*

https://arxiv.org/pdf/1604.07446.pdf

Some demos

http://kevinkdo.com/rrt_demo.html

https://www.youtube.com/watch?v=MT6FyoHefgY

https://www.youtube.com/watch?v=E-IUAL-D9SY

https://www.youtube.com/watch?v=mP4ljdTsvxI

Potential field methods

- Idea [Latombe et al, 1992]
  - Define a potential field
  - Robot moves in the direction of steepest descent on potential function
- Ideally potential function has global minimum at the goal, has no local minima, and is very large around obstacles
- Algorithm outline:
  - place a regular grid over C-space
  - search over the grid with potential function as heuristic

https://www.youtube.com/watch?v=r9FD7P76zJs

Potential field methods

- Pro:
  - Framework can be adapted to any specific scene
- Con:
  - can get stuck in local minima
  - Potential functions that are minima-free are known, but expensive to compute

- Example: RPP (Randomized path planner) is based on potential functions
  - Escapes local minima by executing random walks
  - Succesfully used to
    - performs riveting ops on plane fuselages
    - plan disassembly operations for maintenance of aircraft engines

From UNC: papers + videos

Optimization-based motion planning in dynamic environments (UNC)