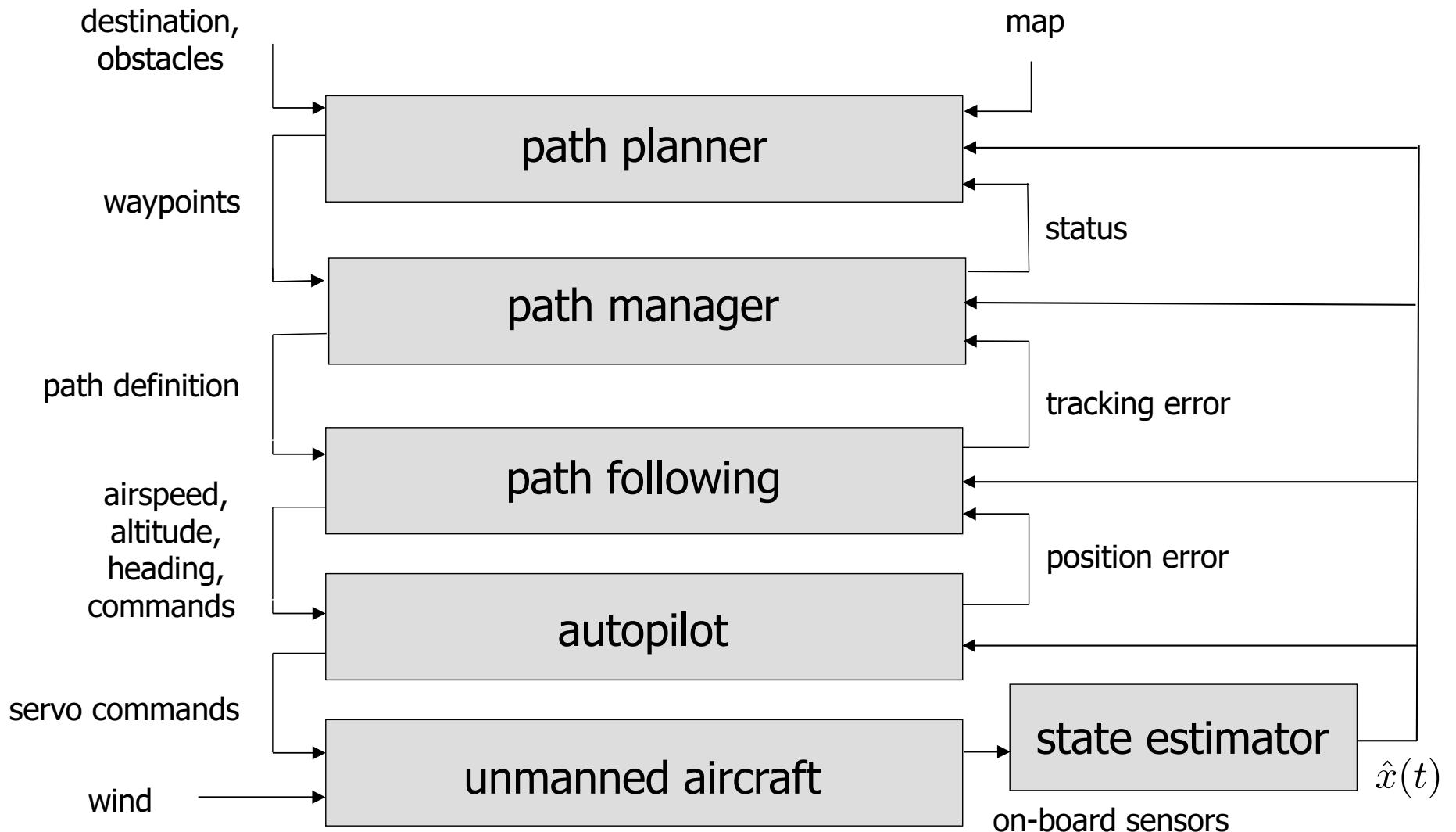




# Chapter 12

## Path Planning

# Control Architecture

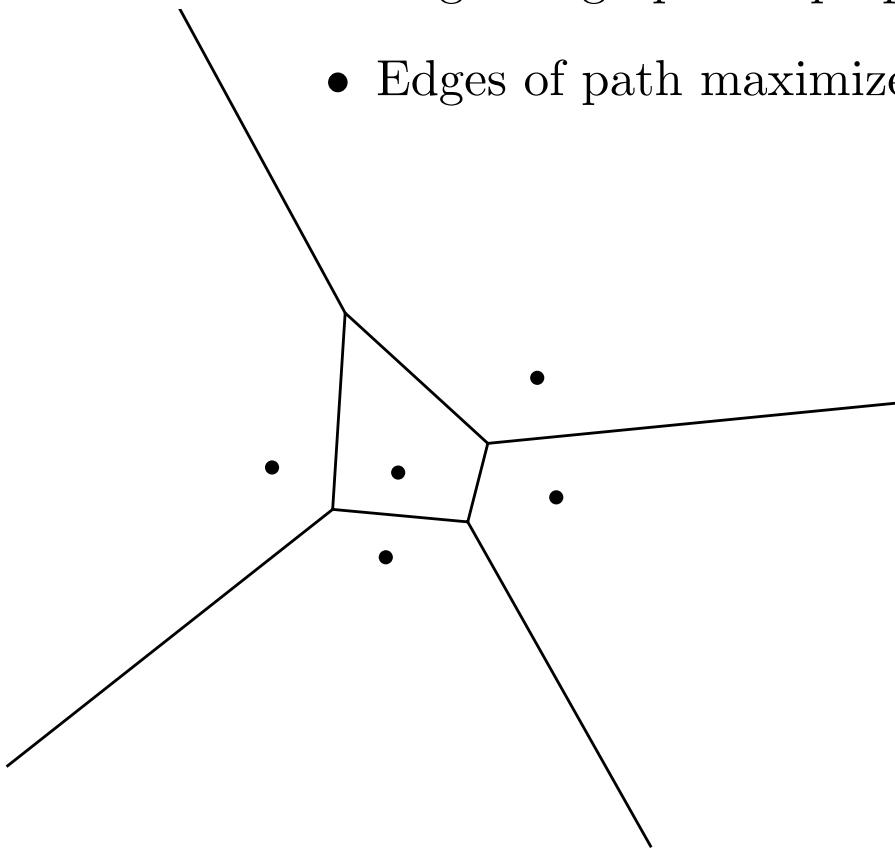


# Path Planning Approaches

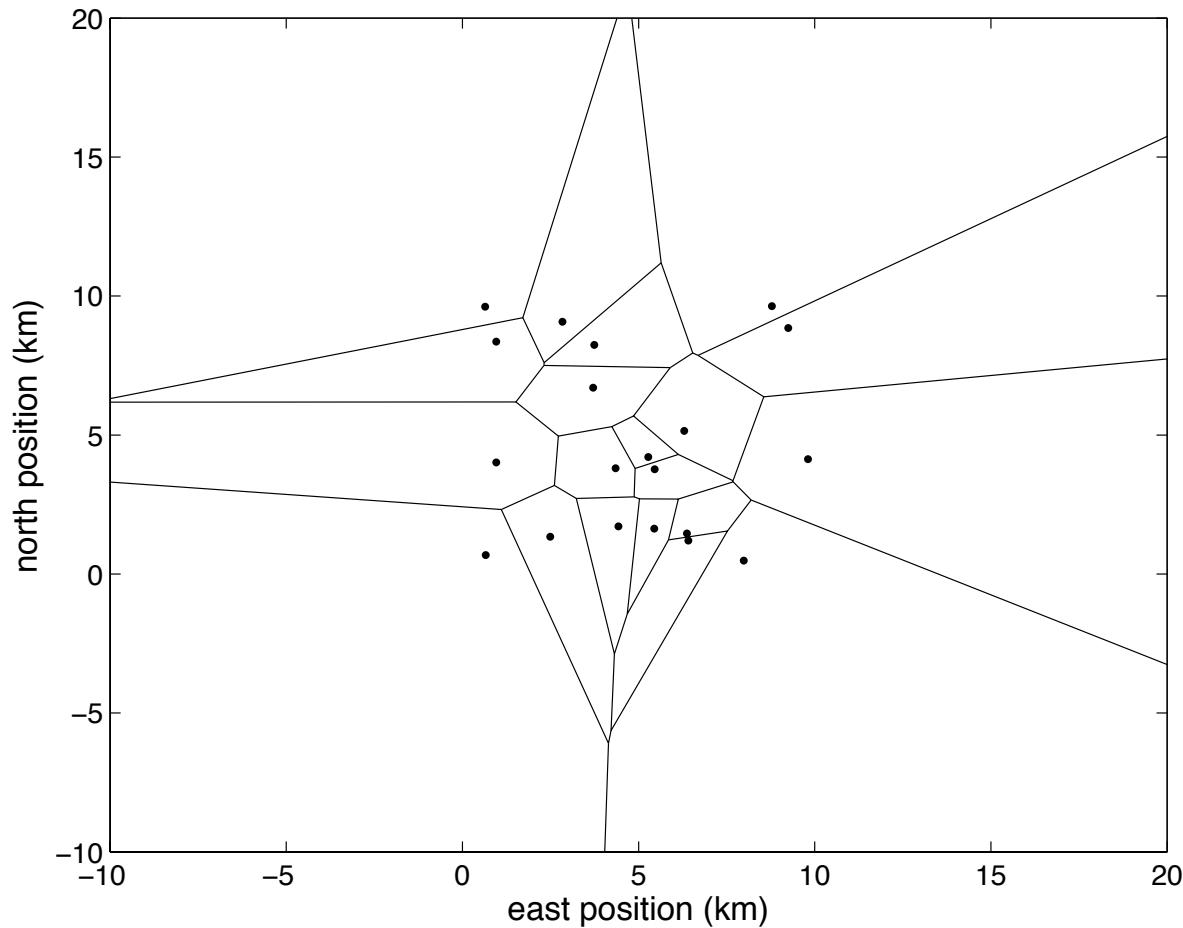
- Deliberative
  - Based on global world knowledge
  - Requires a good map of terrain, obstacles, etc.
  - Can be too computationally intense for dynamic environments
  - Usually executed before the mission
- Reactive
  - Based on what sensors detect on immediate horizon
  - Can respond to dynamic environments
  - Not usually used for entire mission

# Voronoi Graphs

- For finite number  $Q$  point obstacles, Voronoi graph divides search plane into  $Q$  convex cells, each containing one point obstacle
- Interior of cell is closer to its point than any other point in  $Q$
- Edges of graph are perpendicular bisectors between points in  $Q$
- Edges of path maximize distance from point obstacles

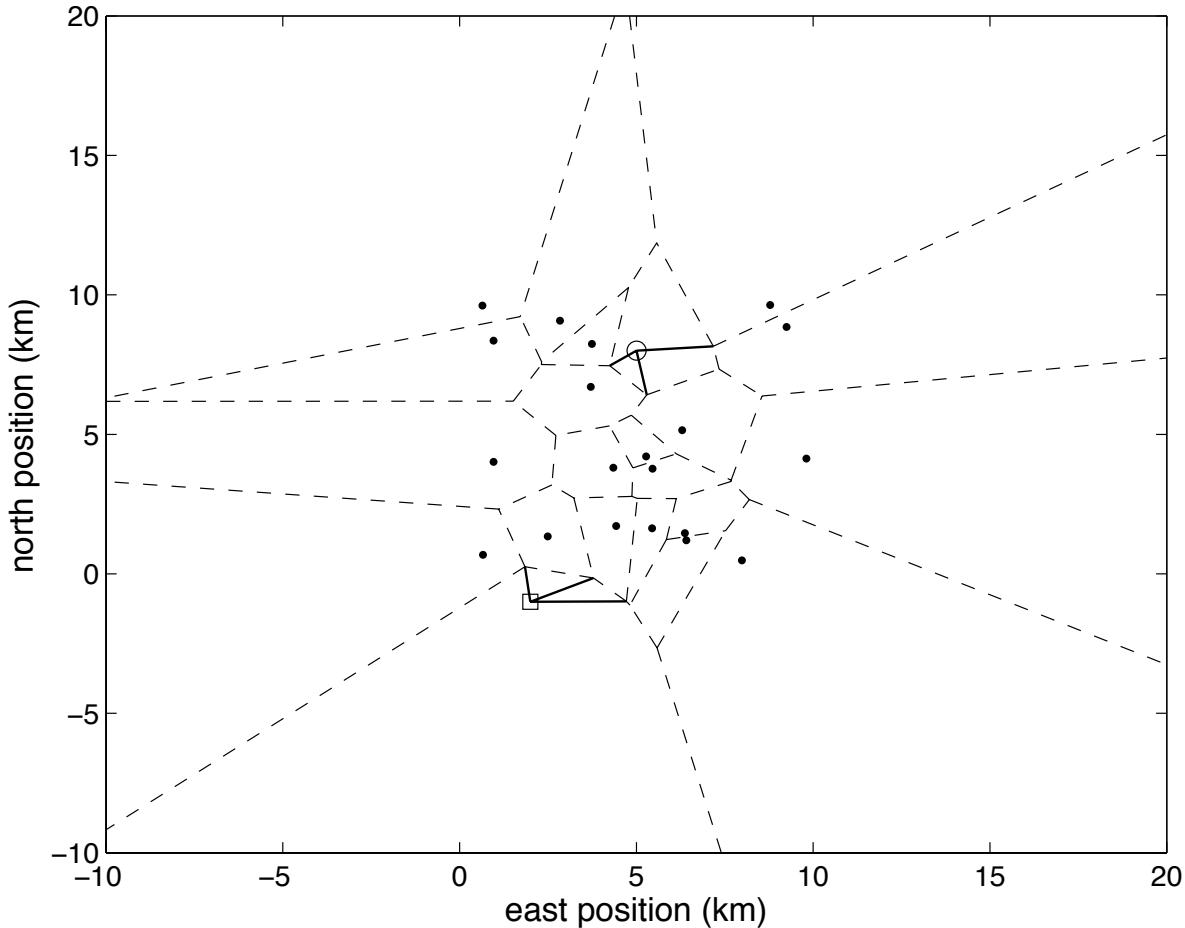


# Voronoi Graph Example



- Graph generated using voronoi command in Matlab
- 20 point obstacles
- Start and end points of path not shown

# Voronoi Graph Example



- Add *start* and *end* points to graph
- Find 3 closest graph nodes to *start* and *end* points
- Add graph edges to *start* and *end* points
- Search graph to find “best” path
- Must define “best”
  - Shortest?
  - Furthest from obstacles?

# Path Cost Calculation

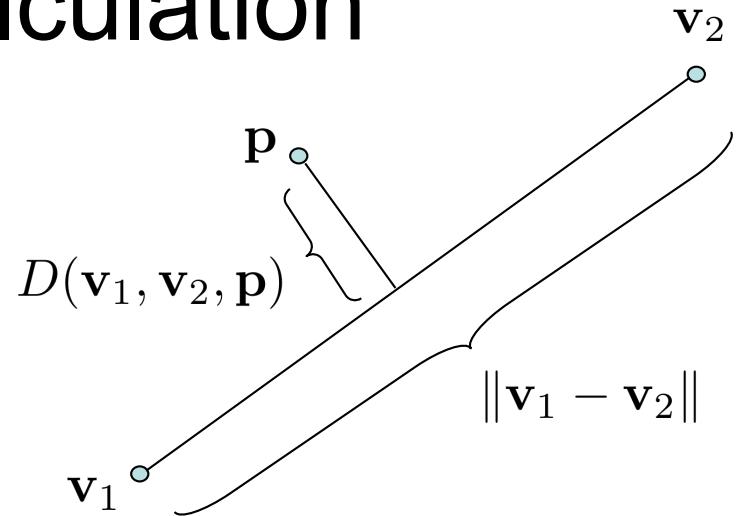
Nodes of graph edge:  $\mathbf{v}_1$  and  $\mathbf{v}_2$   
Point obstacle:  $\mathbf{p}$

Length of edge:  $\|\mathbf{v}_1 - \mathbf{v}_2\|$

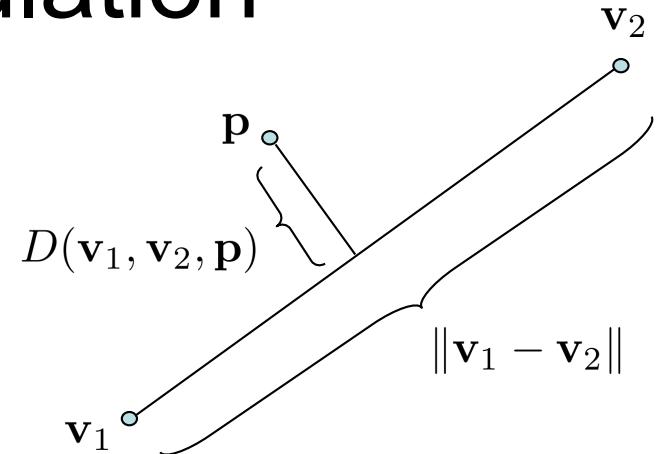
Any point on line segment:  $\mathbf{w}(\sigma) = (1 - \sigma)\mathbf{v}_1 + \sigma\mathbf{v}_2$  where  $\sigma \in [0, 1]$

Minimum distance between  $\mathbf{p}$  and graph edge

$$\begin{aligned} D(\mathbf{v}_1, \mathbf{v}_2, \mathbf{p}) &\triangleq \min_{\sigma \in [0, 1]} \|\mathbf{p} - \mathbf{w}(\sigma)\| \\ &= \min_{\sigma \in [0, 1]} \sqrt{(\mathbf{p} - \mathbf{w}(\sigma))^\top (\mathbf{p} - \mathbf{w}(\sigma))} \\ &= \min_{\sigma \in [0, 1]} \sqrt{\|\mathbf{p} - \mathbf{v}_1\|^2 + 2\sigma(\mathbf{p} - \mathbf{v}_1)^\top (\mathbf{v}_1 - \mathbf{v}_2) + \sigma^2 \|\mathbf{v}_1 - \mathbf{v}_2\|^2}. \end{aligned}$$



# Path Cost Calculation



Value for  $\sigma$  that minimizes  $D$  is

$$\sigma^* = \frac{(\mathbf{v}_1 - \mathbf{p})^\top (\mathbf{v}_1 - \mathbf{v}_2)}{\|\mathbf{v}_1 - \mathbf{v}_2\|^2}$$

Location along edge for which  $D$  is minimum:

$$\mathbf{w}(\sigma^*) = \sqrt{\|\mathbf{p} - \mathbf{v}_1\|^2 - \frac{((\mathbf{v}_1 - \mathbf{p})^\top (\mathbf{v}_1 - \mathbf{v}_2))^2}{\|\mathbf{v}_1 - \mathbf{v}_2\|^2}}$$

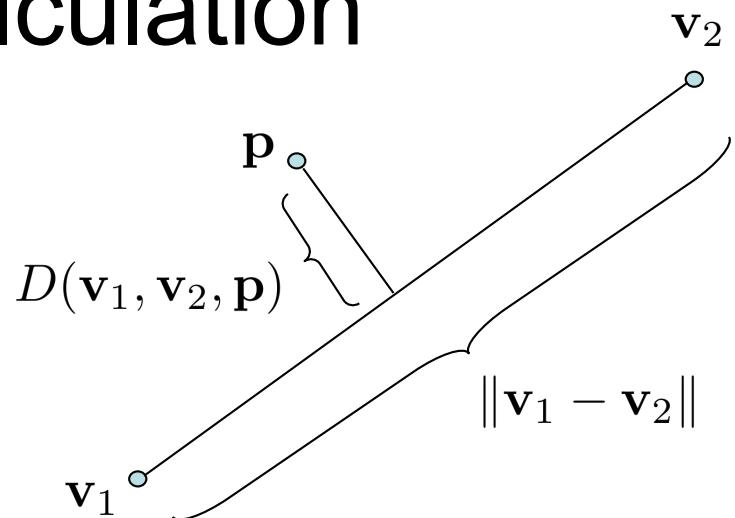
Define distance to edge for  $\sigma^* < 0$ ,  $\sigma^* > 1$ :

$$D'(\mathbf{v}_1, \mathbf{v}_2, \mathbf{p}) \triangleq \begin{cases} \mathbf{w}(\sigma^*) & \text{if } \sigma^* \in [0, 1] \\ \|\mathbf{p} - \mathbf{v}_1\| & \text{if } \sigma^* < 0 \\ \|\mathbf{p} - \mathbf{v}_2\| & \text{if } \sigma^* > 1 \end{cases}$$

Distance between point set  $\mathcal{Q}$  and line segment  $\overline{\mathbf{v}_1 \mathbf{v}_2}$ :

$$D(\mathbf{v}_1, \mathbf{v}_2, \mathcal{Q}) = \min_{\mathbf{p} \in \mathcal{Q}} D'(\mathbf{v}_1, \mathbf{v}_2, \mathbf{p})$$

# Path Cost Calculation



Cost for traveling along edge  $(v_1, v_2)$  is assigned as

$$J(v_1, v_2) = \underbrace{k_1 \|v_1 - v_2\|}_{\text{length of edge}} + \underbrace{\frac{k_2}{D(v_1, v_2, Q)}}_{\text{reciprocal of distance to closest point in } Q}$$

$k_1$  and  $k_2$  are positive weights

Choice of  $k_1$  and  $k_2$  allow tradeoff between path length and proximity to threats.

# Voronoi Path Planning Algorithm

---

**Algorithm 9** Plan Voronoi Path:  $\mathcal{W} = \text{planVoronoi}(\mathcal{Q}, \mathbf{p}_s, \mathbf{p}_e)$

---

**Input:** Obstacle points  $\mathcal{Q}$ , start position  $\mathbf{p}_s$ , end position  $\mathbf{p}_e$

**Require:**  $|\mathcal{Q}| \geq 10$  Randomly add points if necessary.

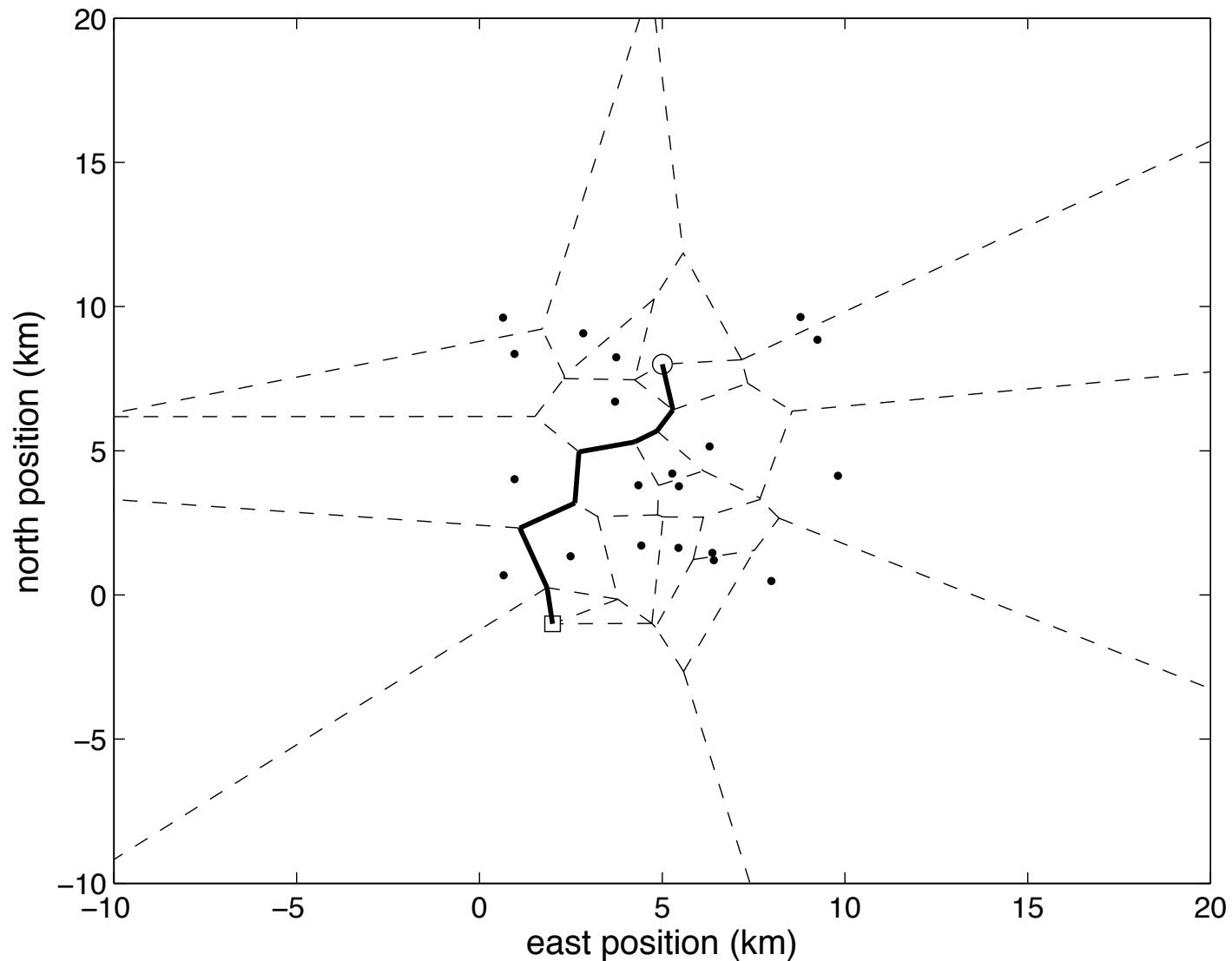
- 1:  $(V, E) = \text{constructVoronoiGraph}(\mathcal{Q})$
  - 2:  $V^+ = V \cup \{\mathbf{p}_s\} \cup \{\mathbf{p}_e\}$
  - 3: Find  $\{\mathbf{v}_{1s}, \mathbf{v}_{2s}, \mathbf{v}_{3s}\}$ , the three closest points in  $V$  to  $\mathbf{p}_s$ , and  
 $\{\mathbf{v}_{1e}, \mathbf{v}_{2e}, \mathbf{v}_{3e}\}$ , the three closest points in  $V$  to  $\mathbf{p}_e$
  - 4:  $E^+ = E \cup_{i=1,2,3} (\mathbf{v}_{is}, \mathbf{p}_s) \cup_{i=1,2,3} (\mathbf{v}_{ie}, \mathbf{p}_e)$
  - 5: **for** Each element  $(\mathbf{v}_a, \mathbf{v}_b) \in E$  **do**
  - 6:     Assign edge cost  $J_{ab} = J(\mathbf{v}_a, \mathbf{v}_b)$  according to equation (12.1).
  - 7: **end for**
  - 8:  $\mathcal{W} = \text{DijkstraSearch}(V^+, E^+, J)$
  - 9: **return**  $\mathcal{W}$
- 

Dijkstra's algorithm is used to search the graph

Voronoi graph and Dijkstra's algorithm code are commonly available  
Matlab:

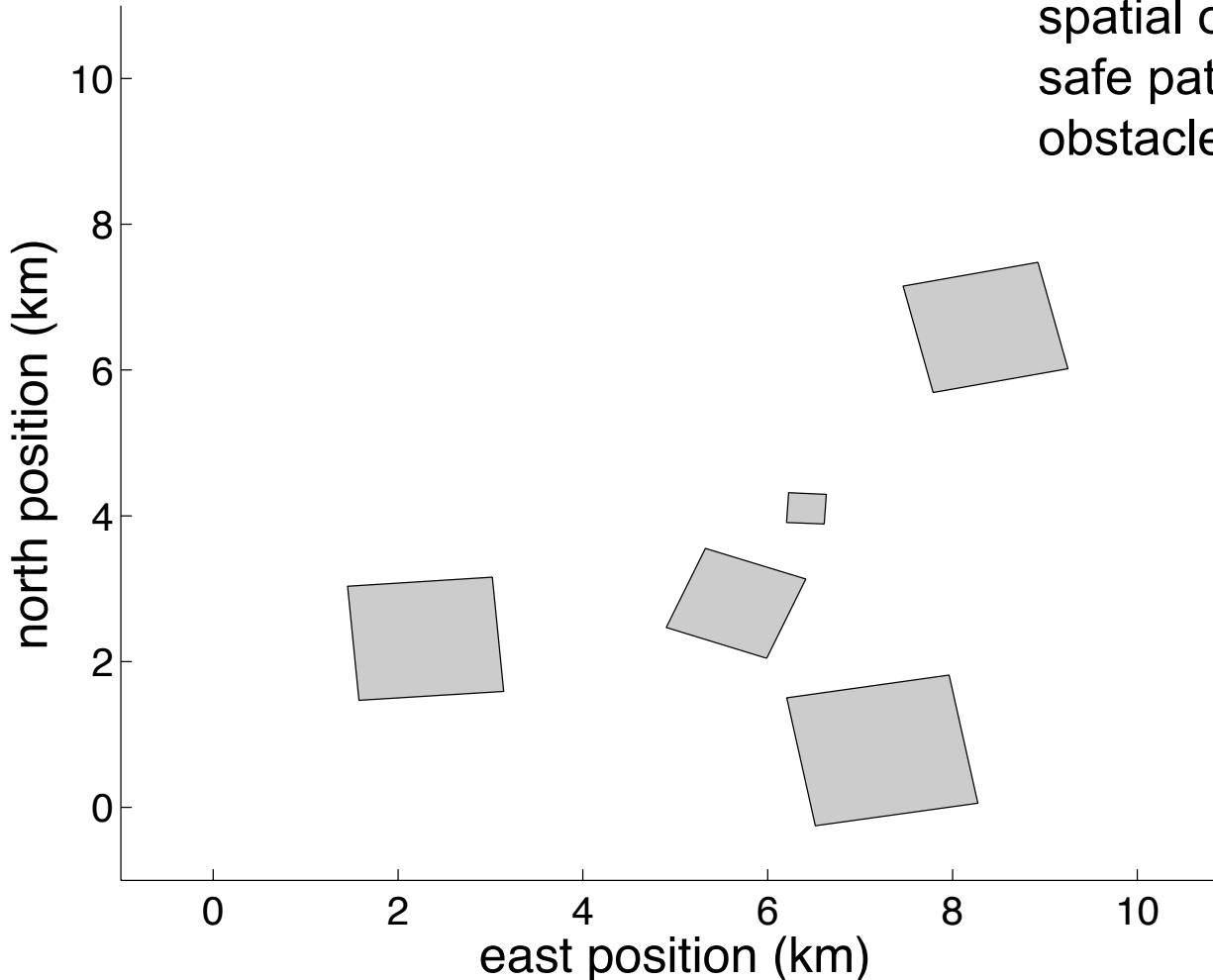
Voronoi -> *voronoi*  
Dijkstra -> *shortestpath*

# Voronoi Path Planning Result



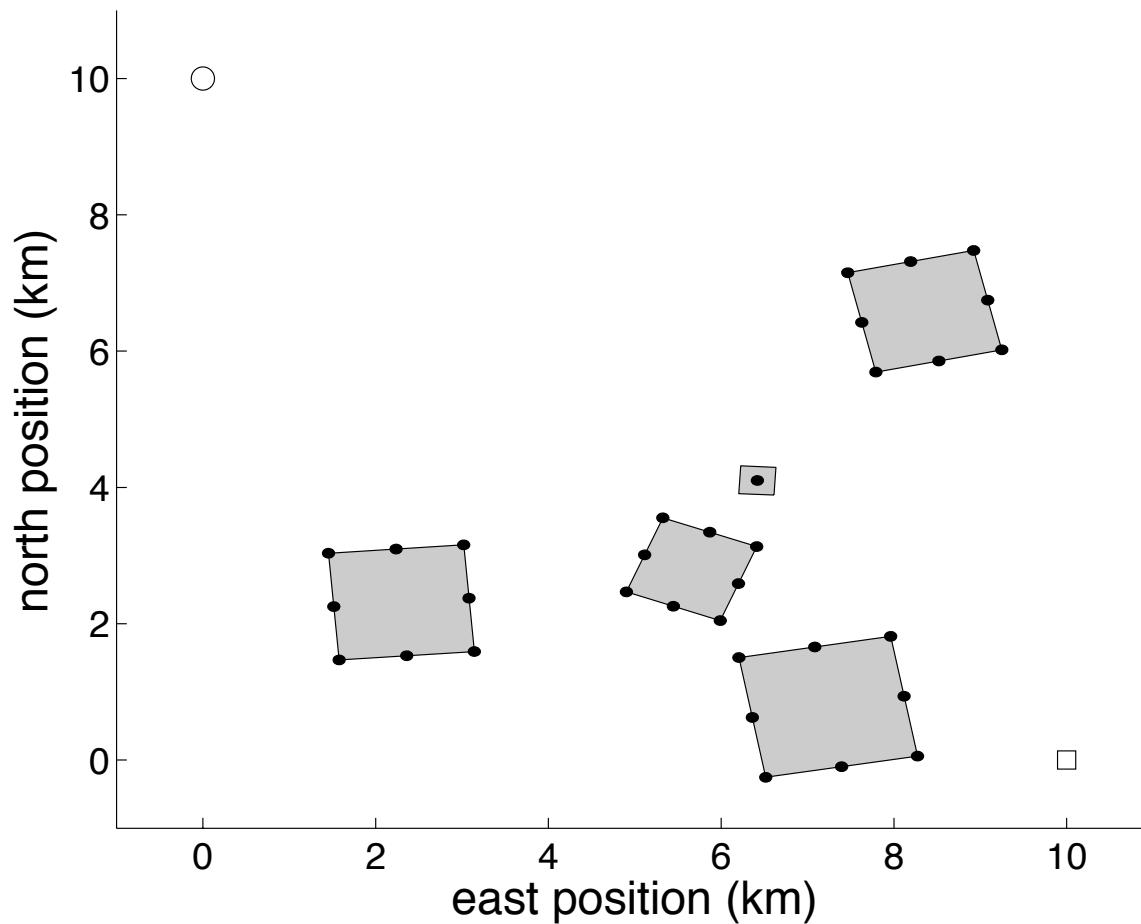
# Voronoi Path Planning – Non-point Obstacles

- How do we handle solid obstacles?
- Point obstacles at center of spatial obstacles won't provide safe path options around obstacles



# Non-point Obstacles – Step 1

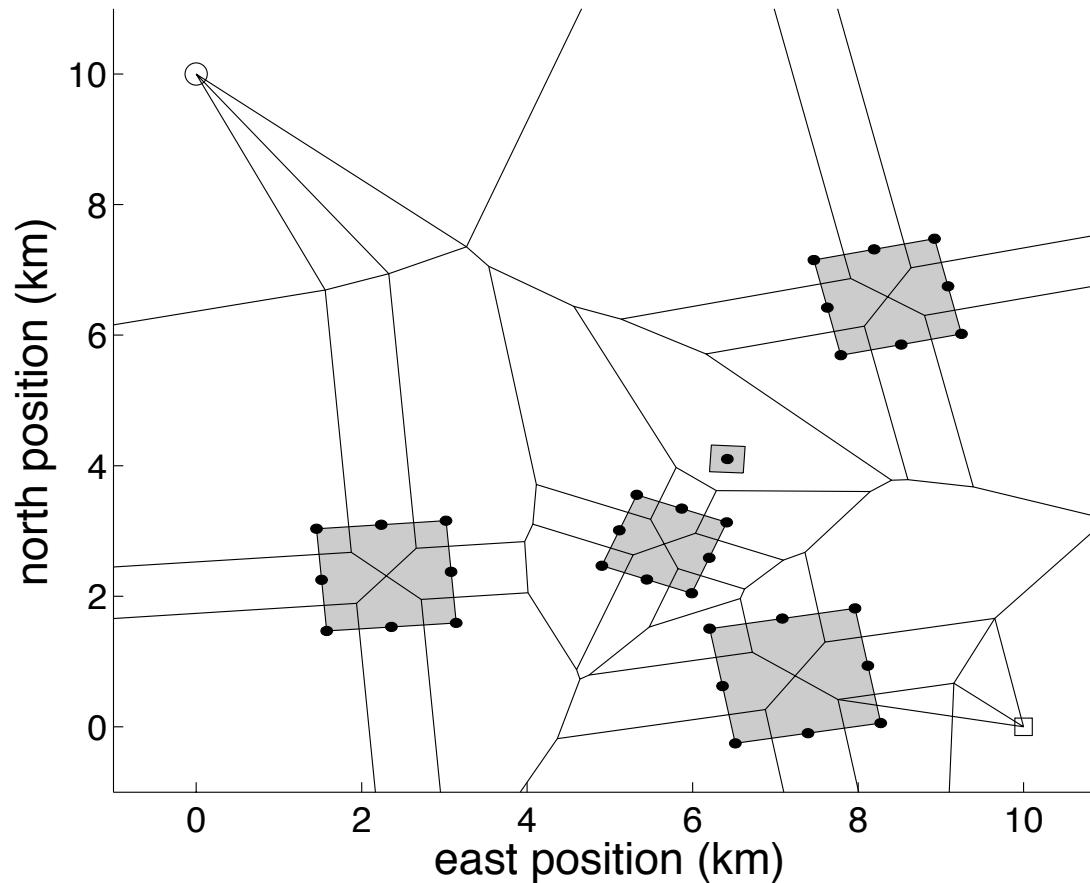
Insert points around perimeter of obstacles



# Non-point Obstacles – Step 2

Construct Voronoi graph

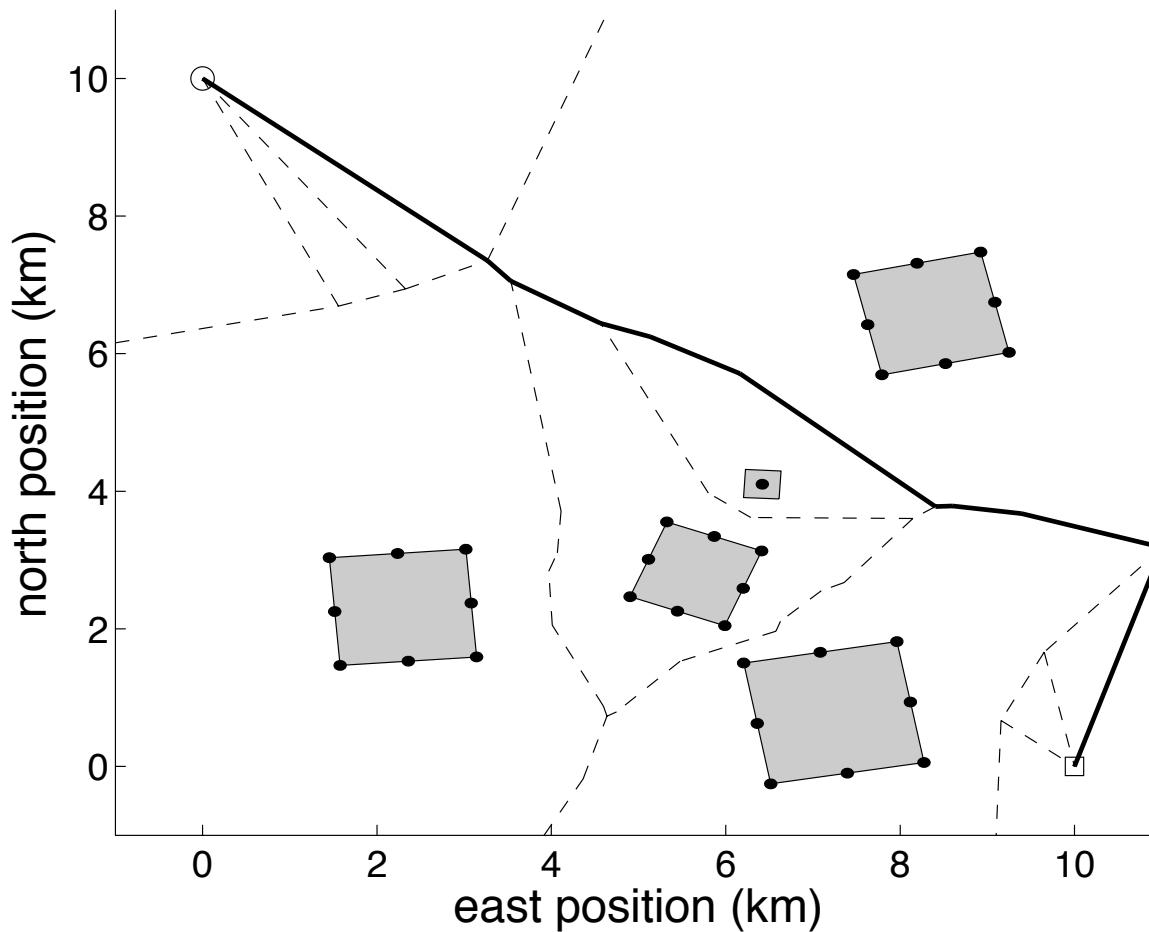
Note infeasible path edges inside obstacles



# Non-point Obstacles – Step 3

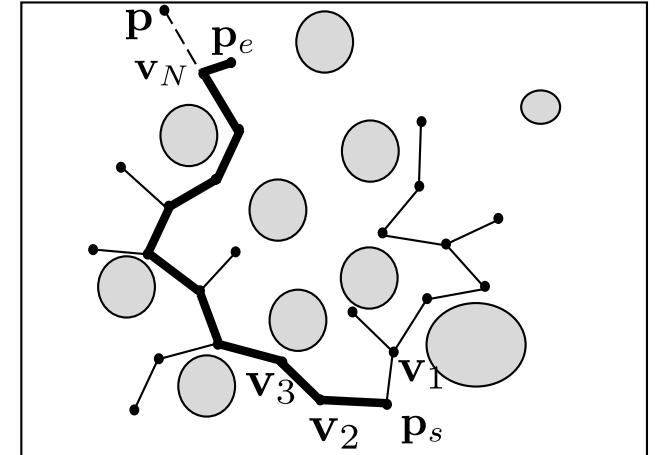
Remove infeasible path edges from graph

Search graph for best path



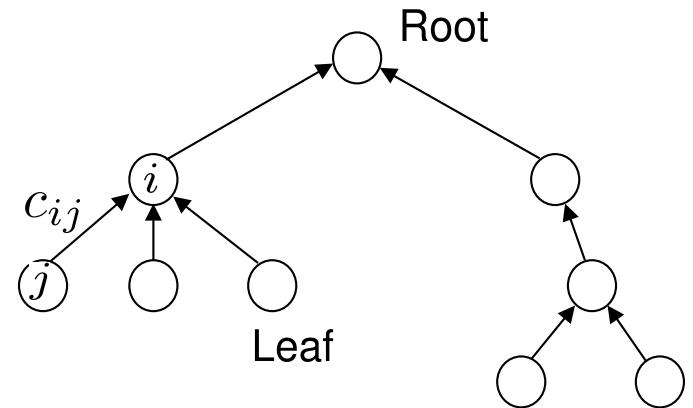
# Rapidly Exploring Random Trees (RRT)

- Exploration algorithm that randomly, but uniformly explores search space
- Can accommodate vehicles with complicated, nonlinear dynamics
- Obstacles represented in a terrain map
- Map can be queried to detect possible collisions
- RRTs can be used to generate a single feasible path or a tree with many feasible paths that can be searched to determine the best one
- If algorithm runs long enough, the optimal path through the terrain will be found



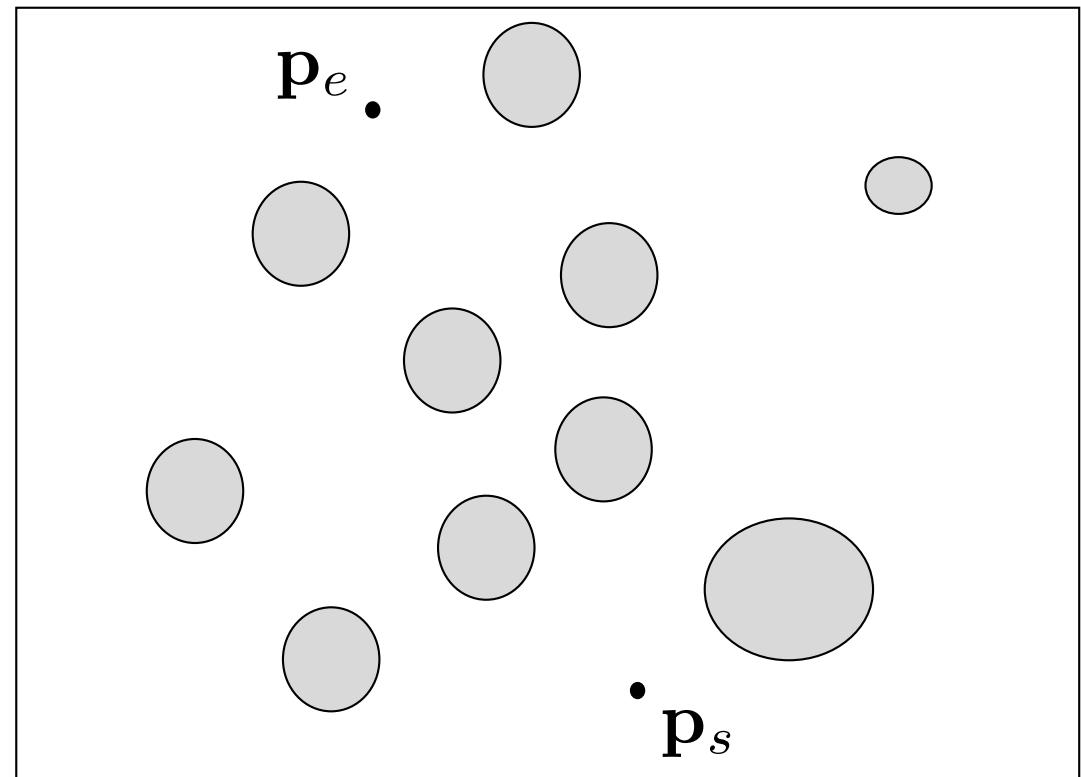
# RRT Tree Structure

- RRT algorithm is implemented using tree data structure
- Tree is special case of directed graph
- Edges are directed from child node to parent
- Every node has one parent, except root
- Nodes represent physical states or configurations
- Edges represent feasible paths between states
- Each edge has cost associated traversing feasible path between states



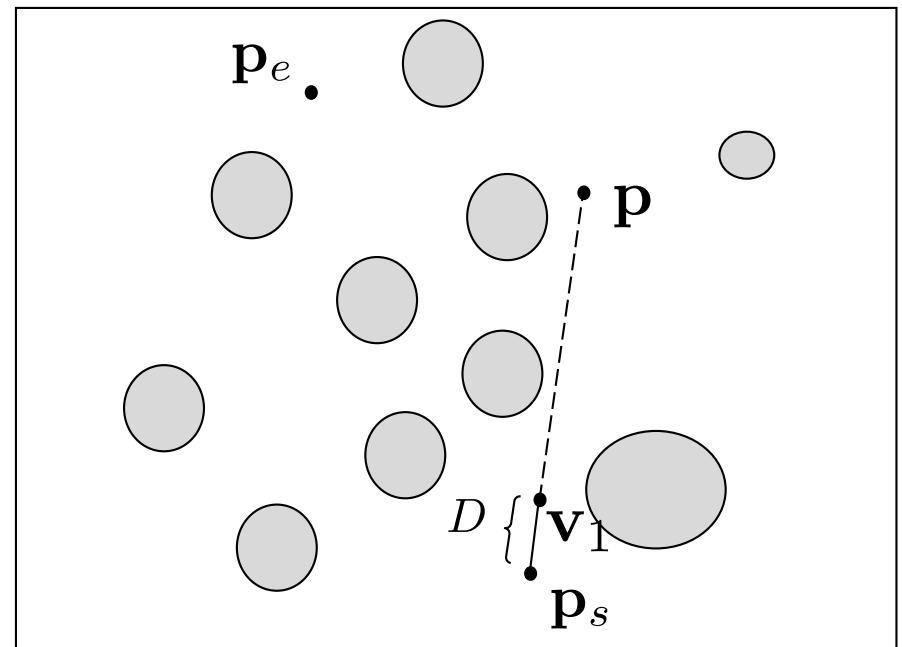
# RRT Algorithm

- Algorithm initialized
  - start node
  - end node
  - terrain/obstacle map



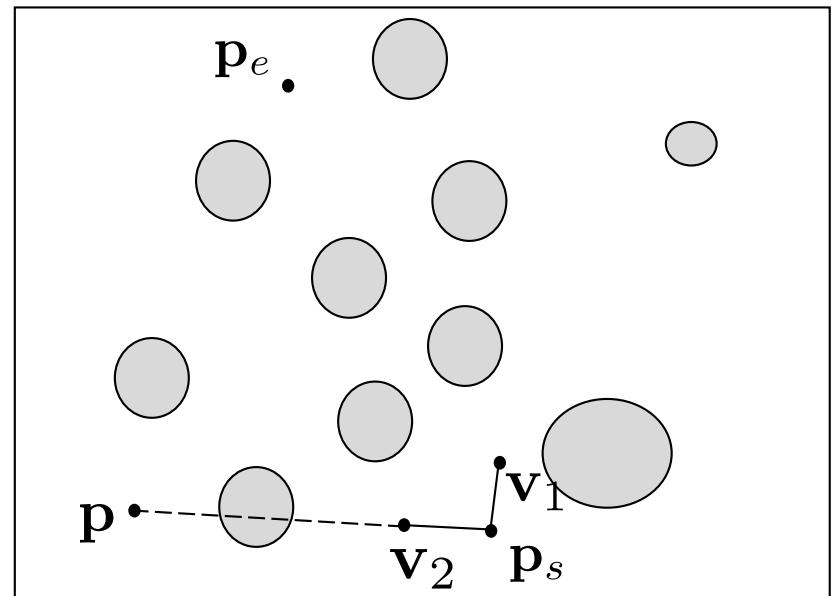
# RRT Algorithm

- Randomly select configuration  $\mathbf{p}$  in workspace
- Select new configuration  $\mathbf{v}_1$  fixed distance  $D$  from start point along line connecting  $\mathbf{p}_s$  and  $\mathbf{p}$
- Check new segment for collisions
- If collision-free, insert  $\mathbf{v}_1$  into tree.



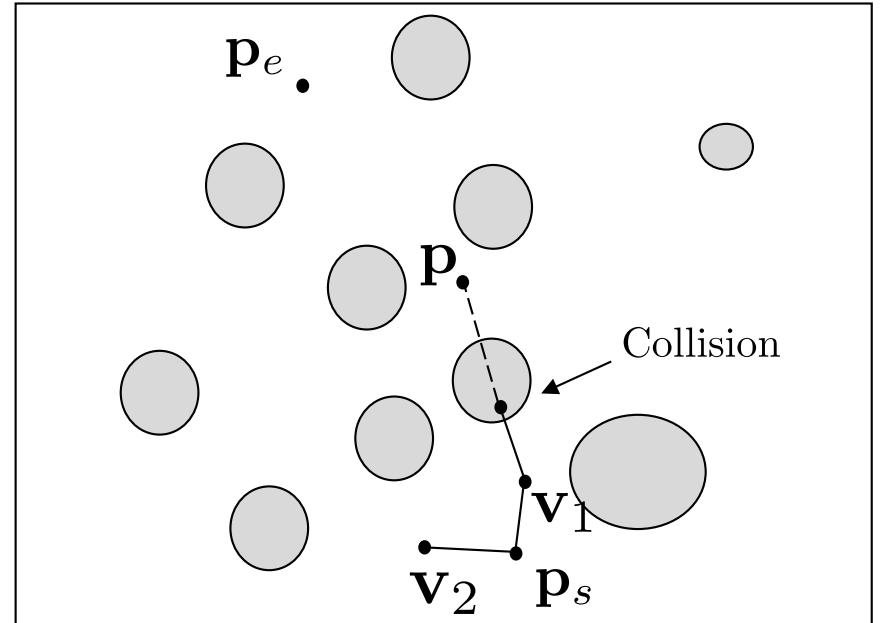
# RRT Algorithm

- Generate random configuration  $\mathbf{p}$  in workspace
- Search tree to find node closest to  $\mathbf{p}$
- From node closest to  $\mathbf{p}$ , move distance  $D$  along line connecting node and  $\mathbf{p}$  to establish configuration  $\mathbf{v}_2$
- Check new segment for collisions
- If collision-free, insert  $\mathbf{v}_2$  into tree



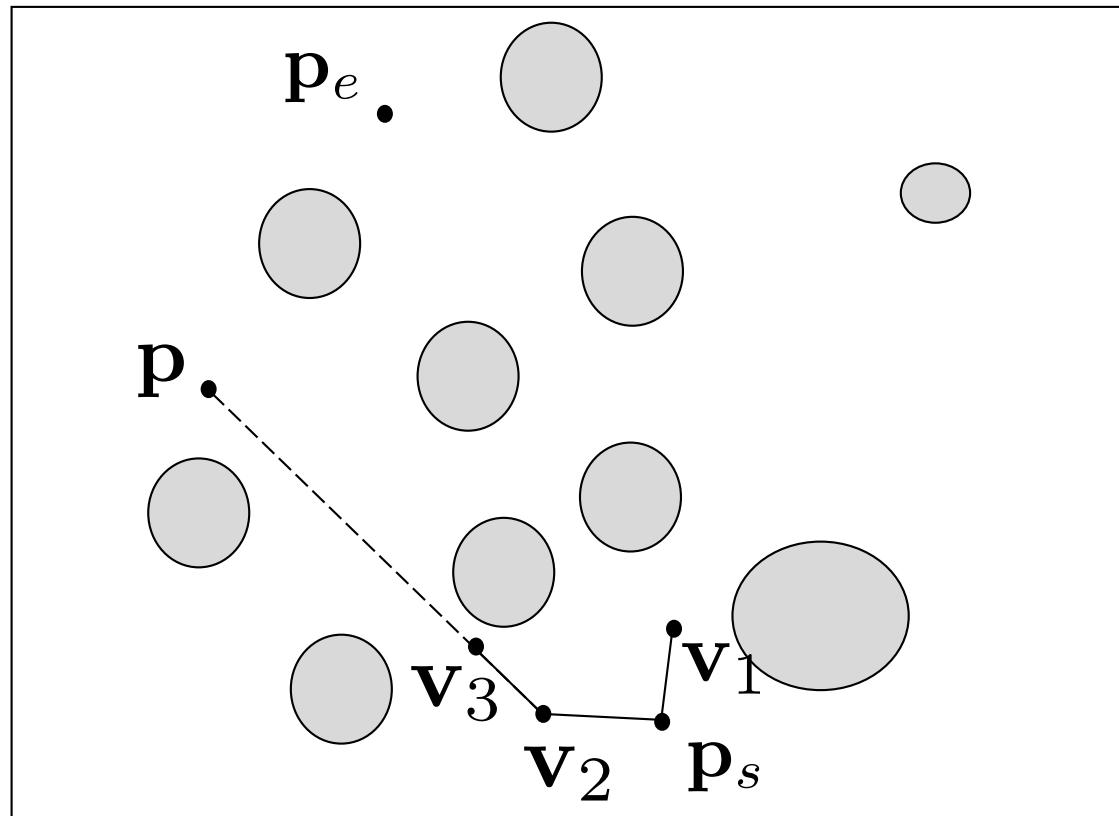
# RRT Algorithm

- Generate random configuration  $\mathbf{p}$  in workspace
- Search tree to find node closest to  $\mathbf{p}$
- From node closest to  $\mathbf{p}$ , move distance  $D$  along line connecting node and  $\mathbf{p}$  to establish new node
- Check new segment for collisions
- If collision-free, insert new node into tree



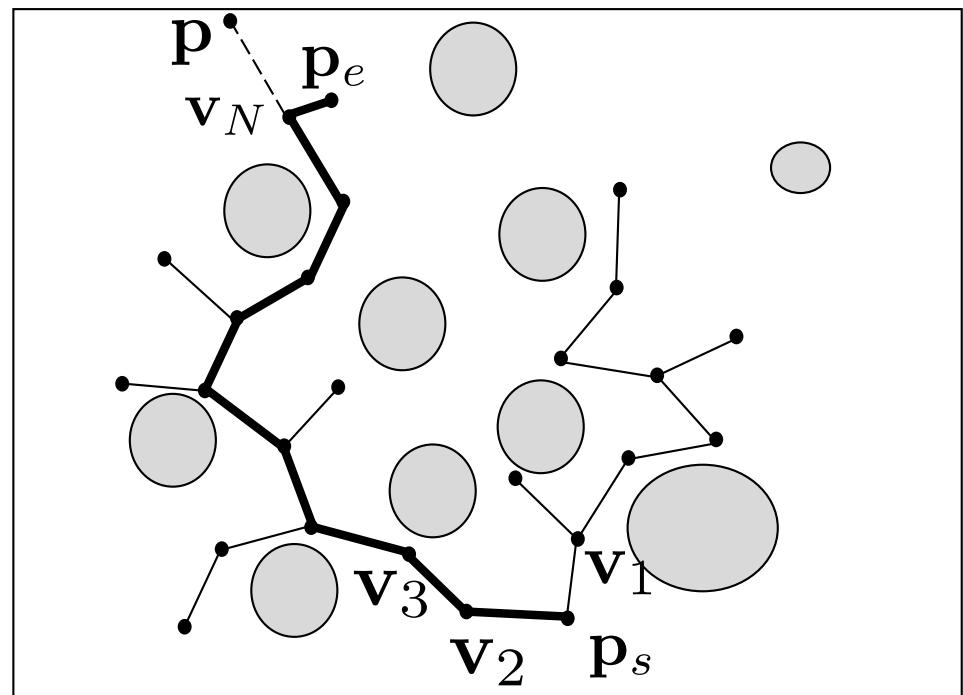
# RRT Algorithm

- Continue adding nodes and checking for collisions



# RRT Algorithm

- Continue until node is generated that is within distance  $D$  of end node
- At this point, terminate algorithm or search for additional feasible paths



# RRT Algorithm

---

**Algorithm 10** Plan RRT Path:  $\mathcal{W} = \text{planRRT}(\mathcal{T}, \mathbf{p}_s, \mathbf{p}_e)$

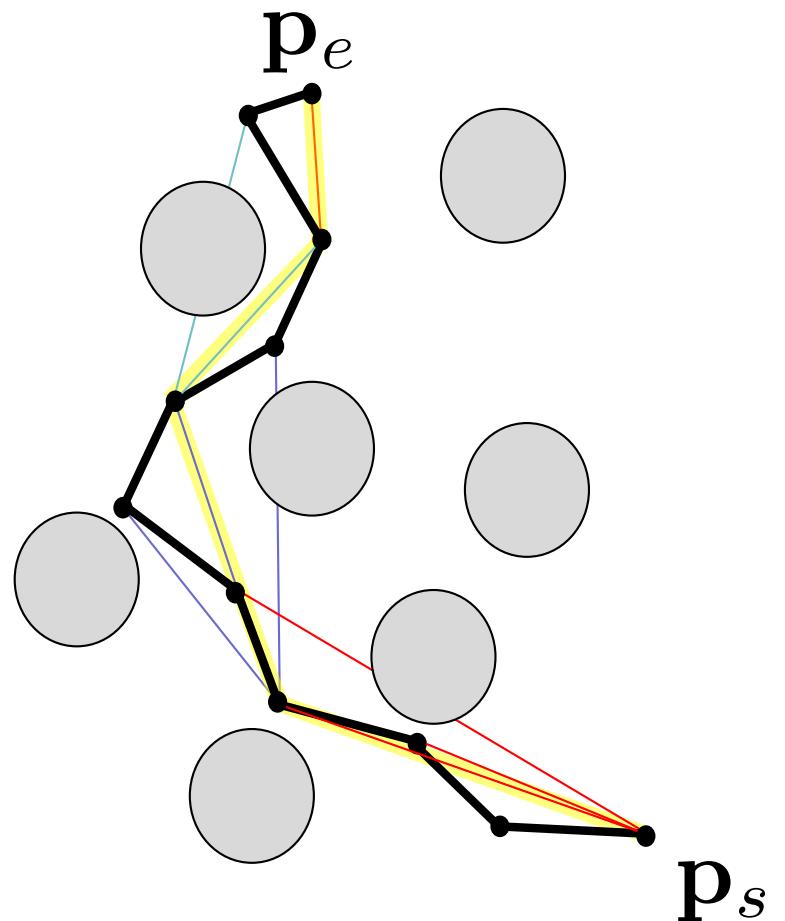
---

**Input:** Terrain map  $\mathcal{T}$ , start configuration  $\mathbf{p}_s$ , end configuration  $\mathbf{p}_e$

- 1: Initialize RRT graph  $G = (V, E)$  as  $V = \{\mathbf{p}_s\}$ ,  $E = \emptyset$
  - 2: **while** The end node  $\mathbf{p}_e$  is not connected to  $G$ , i.e.,  $\mathbf{p}_e \notin V$  **do**
  - 3:    $\mathbf{p} \leftarrow \text{generateRandomConfiguration}(\mathcal{T})$
  - 4:    $\mathbf{v}^* \leftarrow \text{findClosestConfiguration}(\mathbf{p}, V)$
  - 5:    $\mathbf{v}^+ \leftarrow \text{planPath}(\mathbf{v}^*, \mathbf{p}, D)$
  - 6:   **if**  $\text{existFeasiblePath}(\mathcal{T}, \mathbf{v}^*, \mathbf{v}^+)$  **then**
  - 7:     Update graph  $G = (V, E)$  as  $V \leftarrow V \cup \{\mathbf{v}^+\}$ ,  $E \leftarrow E \cup \{(\mathbf{v}^*, \mathbf{v}^+)\}$
  - 8:     Update edge costs as  $C[(\mathbf{v}^*, \mathbf{v}^+)] \leftarrow \text{pathLength}(\mathbf{v}^*, \mathbf{v}^+)$
  - 9:   **end if**
  - 10:   **if**  $\text{existFeasiblePath}(\mathcal{T}, \mathbf{v}^+, \mathbf{p}_e)$  **then**
  - 11:     Update graph  $G = (V, E)$  as  $V \leftarrow V \cup \{\mathbf{p}_e\}$ ,  $E \leftarrow E \cup \{(\mathbf{v}^*, \mathbf{p}_e)\}$
  - 12:     Update edge costs as  $C[(\mathbf{v}^*, \mathbf{p}_e)] \leftarrow \text{pathLength}(\mathbf{v}^*, \mathbf{p}_e)$
  - 13:   **end if**
  - 14: **end while**
  - 15:  $\mathcal{W} = \text{findShortestPath}(G, C).$
  - 16: **return**  $\mathcal{W}$
-

# Path Smoothing

- Start with initial point (1)
- Make connections to subsequent points in path (2), (3), (4) ...
- When connection collides with obstacle, add previous waypoint to smoothed path
- Continue smoothing from this point to end of path



# Path Smoothing Algorithm

---

**Algorithm 11** Smooth RRT Path:  $(\mathcal{W}_s, C_s) = \text{smoothRRT}(\mathcal{T}, \mathcal{W}, C)$

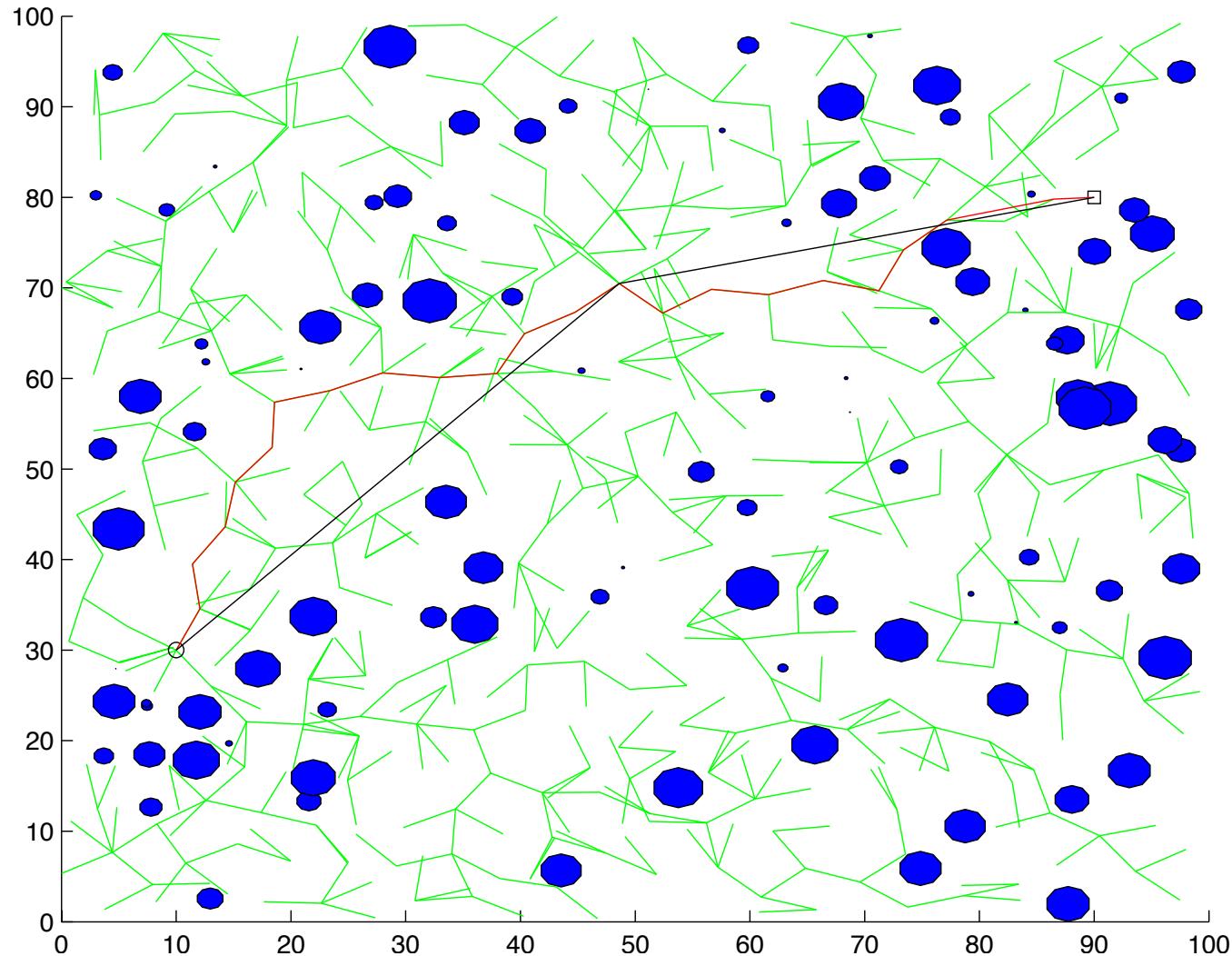
---

**Input:** Terrain map  $\mathcal{T}$ , waypoint path  $\mathcal{W} = \{\mathbf{w}_1, \dots, \mathbf{w}_N\}$ , cost matrix  $C$

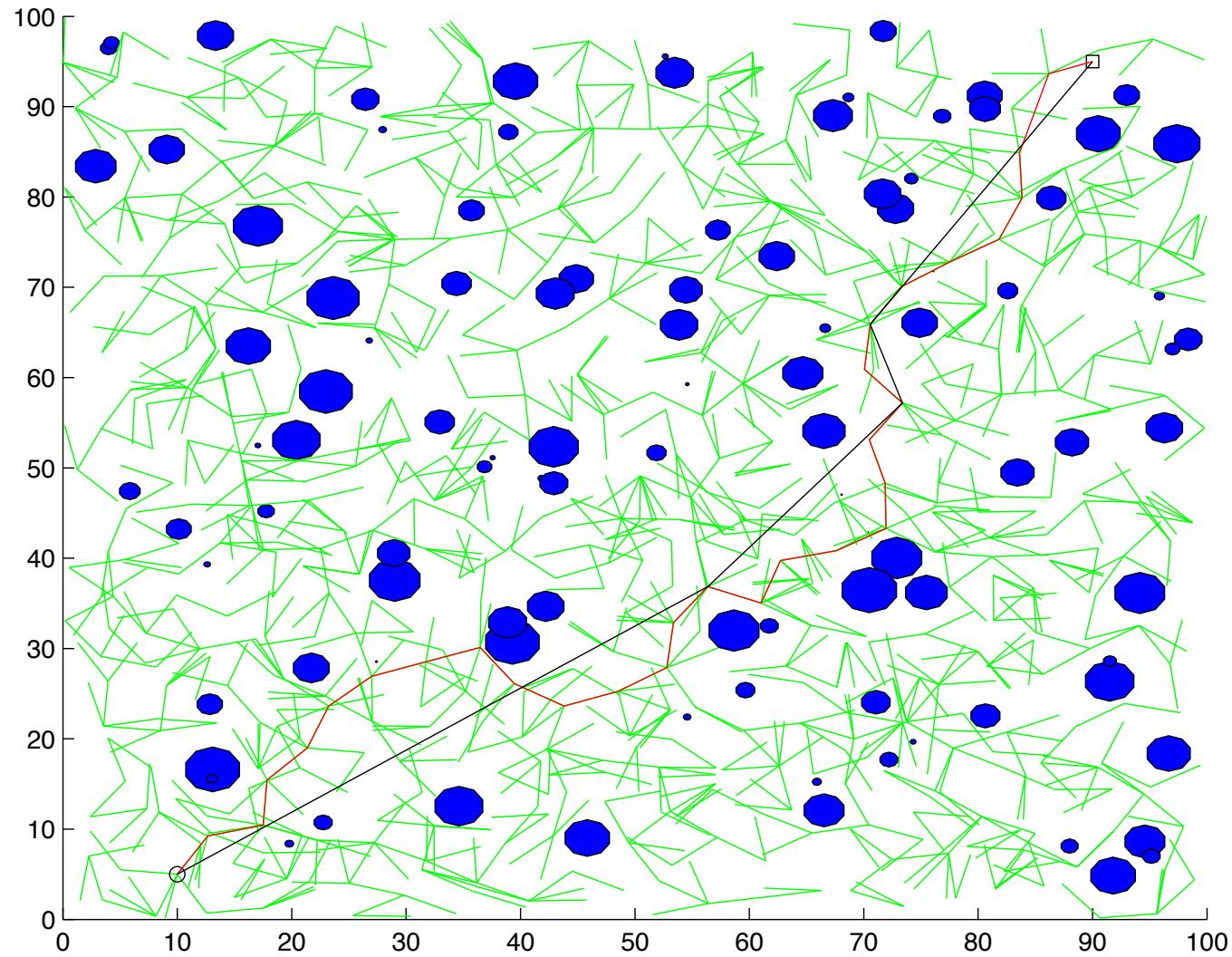
```
1: Initialized smoothed path  $\mathcal{W}_s \leftarrow \{\mathbf{w}_1\}$ 
2: Initialize pointer to current node in  $\mathcal{W}_s$ :  $i \leftarrow 1$ 
3: Initialize pointer to next node in  $\mathcal{W}$ :  $j \leftarrow 2$ 
4: while  $j < N$  do
5:    $\mathbf{w}_s \leftarrow \text{getNode}(\mathcal{W}_s, i)$ 
6:    $\mathbf{w}^+ \leftarrow \text{getNode}(\mathcal{W}, j + 1)$ 
7:   if  $\text{existFeasiblePath}(\mathcal{T}, \mathbf{w}_s, \mathbf{w}^+) = \text{FALSE}$  then
8:     Get last node:  $\mathbf{w} \leftarrow \text{getNode}(\mathcal{W}, j)$ 
9:     Add deconflicted node to smoothed path:  $\mathcal{W}_s \leftarrow \mathcal{W}_s \cup \{\mathbf{w}\}$ 
10:    Update smoothed cost:  $C_s[(\mathbf{w}_s, \mathbf{w})] \leftarrow \text{pathLength}(\mathbf{w}_s, \mathbf{w})$ 
11:     $i \leftarrow i + 1$   $i \leftarrow j$ 
12:   end if
13:    $j \leftarrow j + 1$ 
14: end while
15: Add last node from  $\mathcal{W}$ :  $\mathcal{W}_s \leftarrow \mathcal{W}_s \cup \{\mathbf{w}_N\}$ 
16: Update smoothed cost:  $C_s[(\mathbf{w}_i, \mathbf{w}_N)] \leftarrow \text{pathLength}(\mathbf{w}_i, \mathbf{w}_N)$ 
17: return  $\mathcal{W}_s$ 
```

---

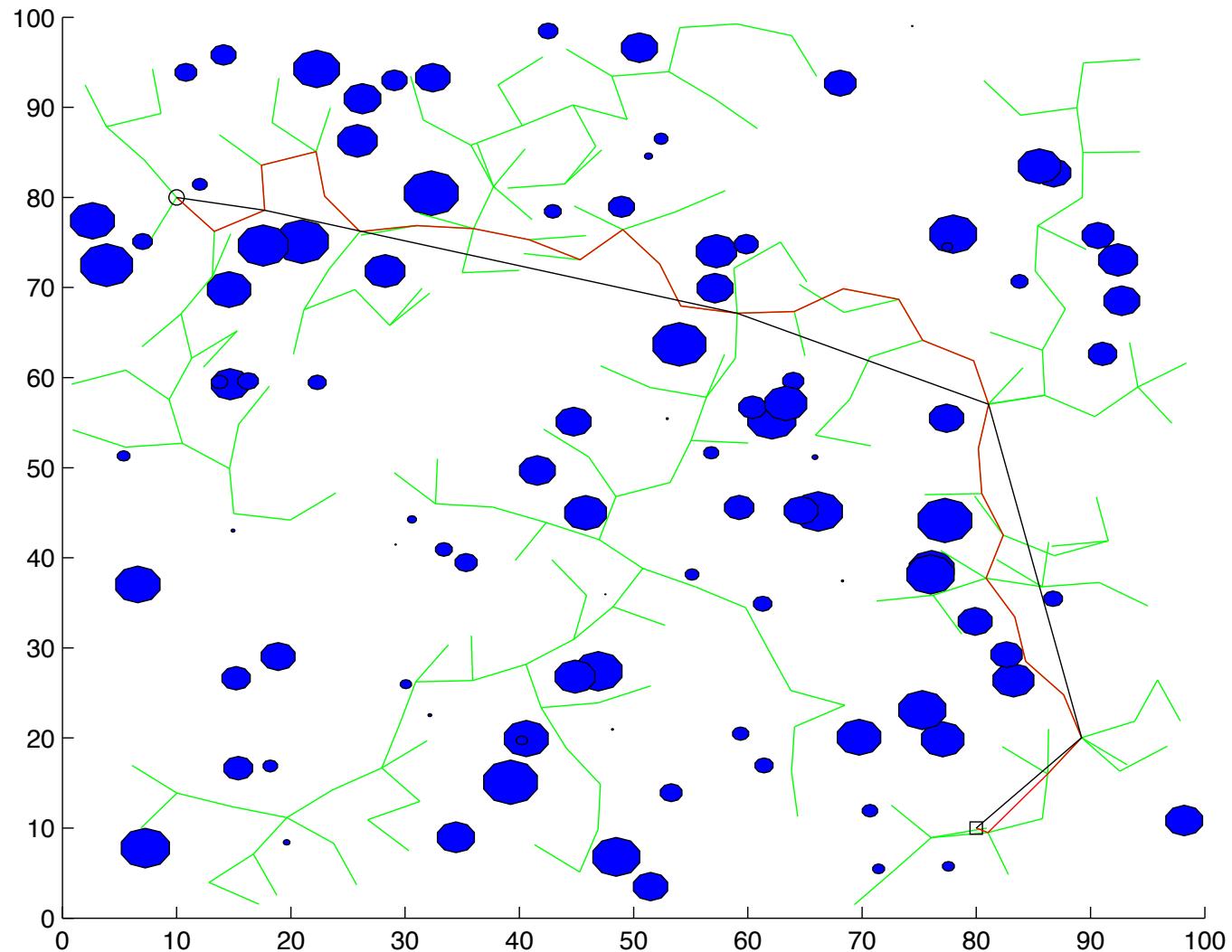
# RRT Results



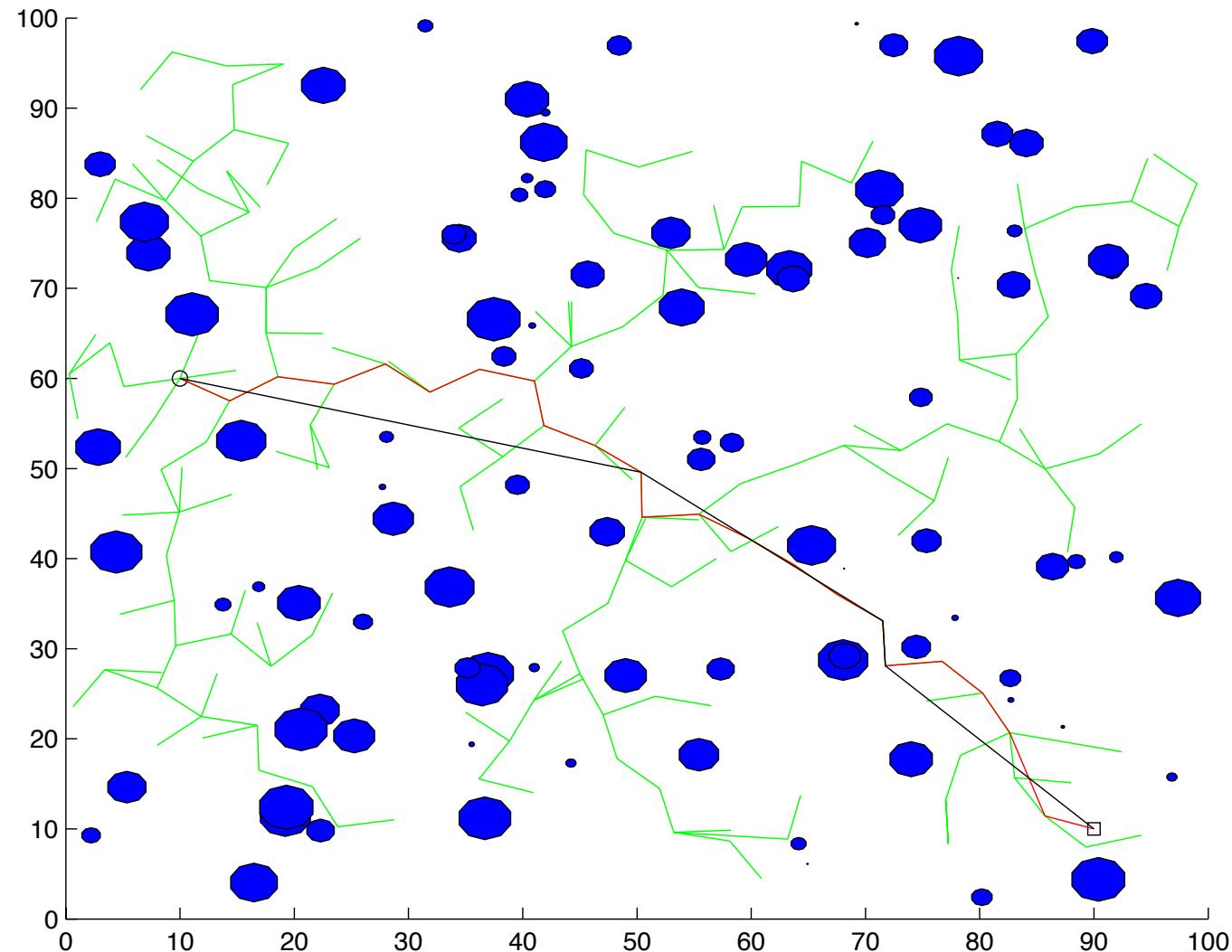
# RRT Results



# RRT Results

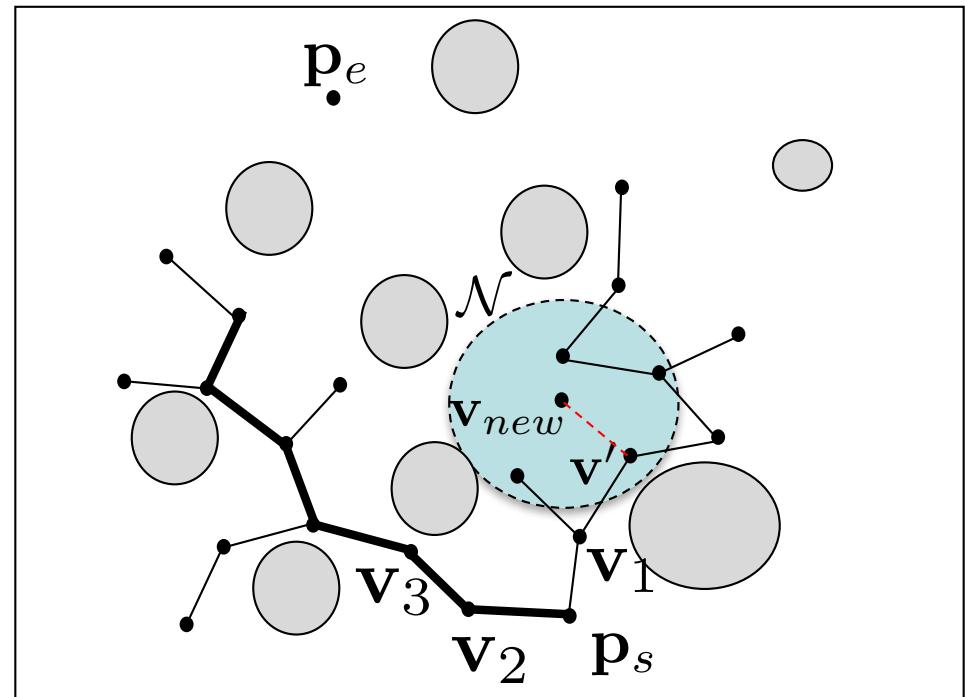


# RRT Results



# RRT\* Algorithm – Extend Step

- Generate new potential node  $\mathbf{v}_{new}$  identical to RRT.
- Instead of finding the closest node in the tree, find all nodes within neighborhood  $\mathcal{N}$ .
- Let  $C(\mathbf{v})$  be the path cost from  $\mathbf{p}_s$  to  $\mathbf{v}$ , and let  $c(\mathbf{v}_1, \mathbf{v}_2)$  be the path cost from  $\mathbf{v}_1$  to  $\mathbf{v}_2$ .
- Let  $\mathbf{v}' = \arg \min_{\mathbf{v} \in \mathcal{N}} (C(\mathbf{v}) + c(\mathbf{v}_{new}, \mathbf{v}))$  (i.e., closest node in  $\mathcal{N}$  to  $\mathbf{v}_{new}$ ).
- Add node  $V \leftarrow V \cup \{\mathbf{v}_{new}\}$ .
- Add edge  $E \leftarrow E \cup \{(\mathbf{v}', \mathbf{v}_{new})\}$ .
- Set  $C(\mathbf{v}_{new}) = C(\mathbf{v}') + c(\mathbf{v}_{new}, \mathbf{v})$ .



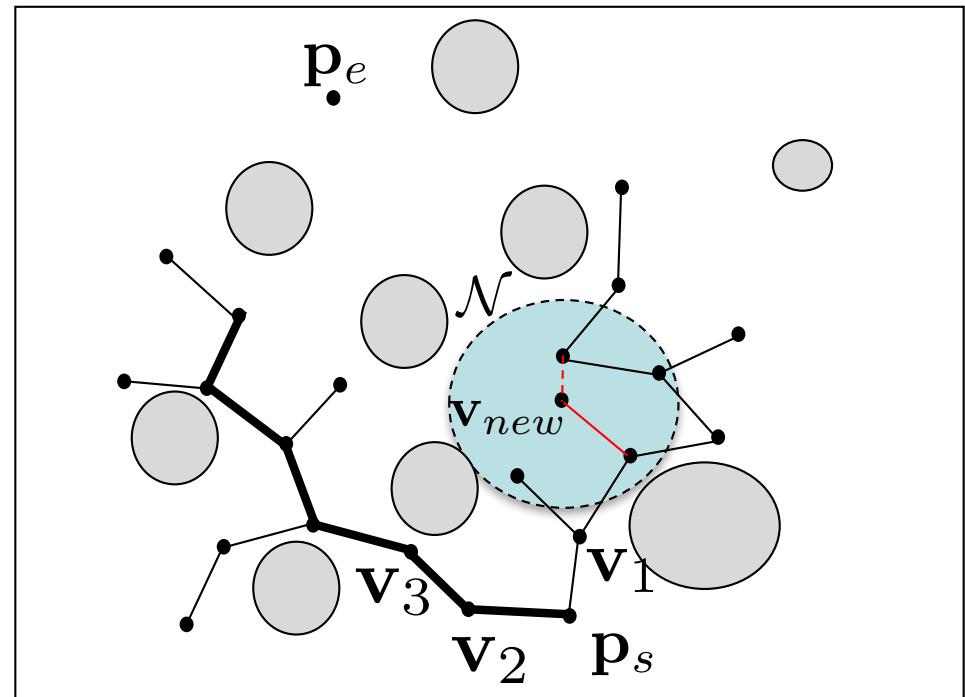
# RRT\* Algorithm – Re-wire Step

- Check all nodes in  $\mathbf{v} \in \mathcal{N}$  to see if

$$C(\mathbf{v}_{new}) + c(\mathbf{v}_{new}, \mathbf{v}) < C(\mathbf{v})$$

i.e., if re-routing through  $\mathbf{v}_{new}$  reduces the path length.

- If so, remove edge between  $\mathbf{v}$  and its parent, and add edge between  $\mathbf{v}$  and  $\mathbf{v}_{new}$ .



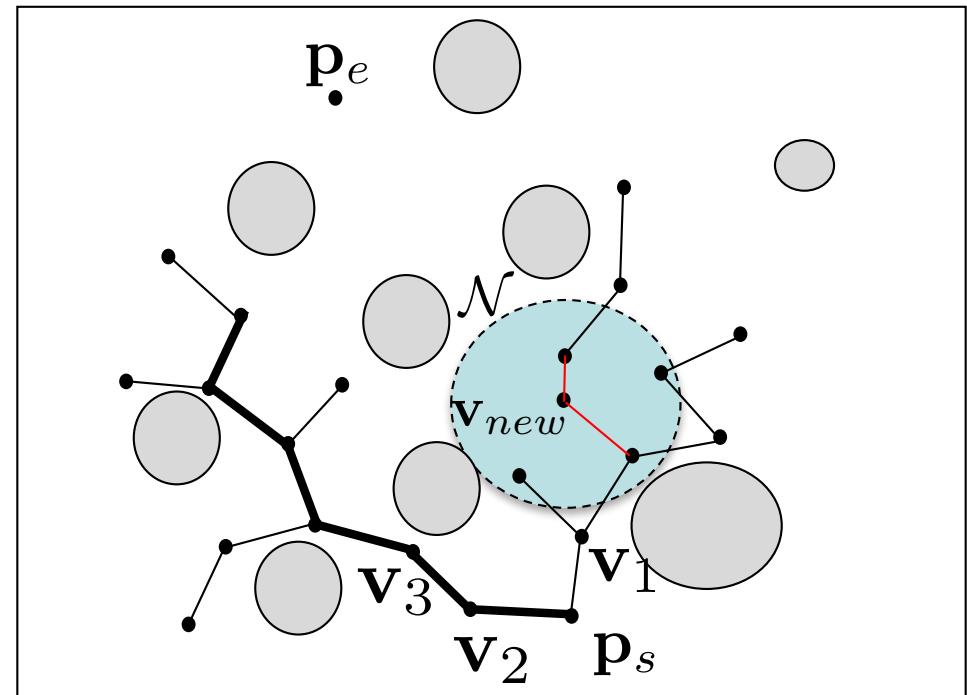
# RRT\* Algorithm – Re-wire Step

- Check all nodes in  $\mathbf{v} \in \mathcal{N}$  to see if

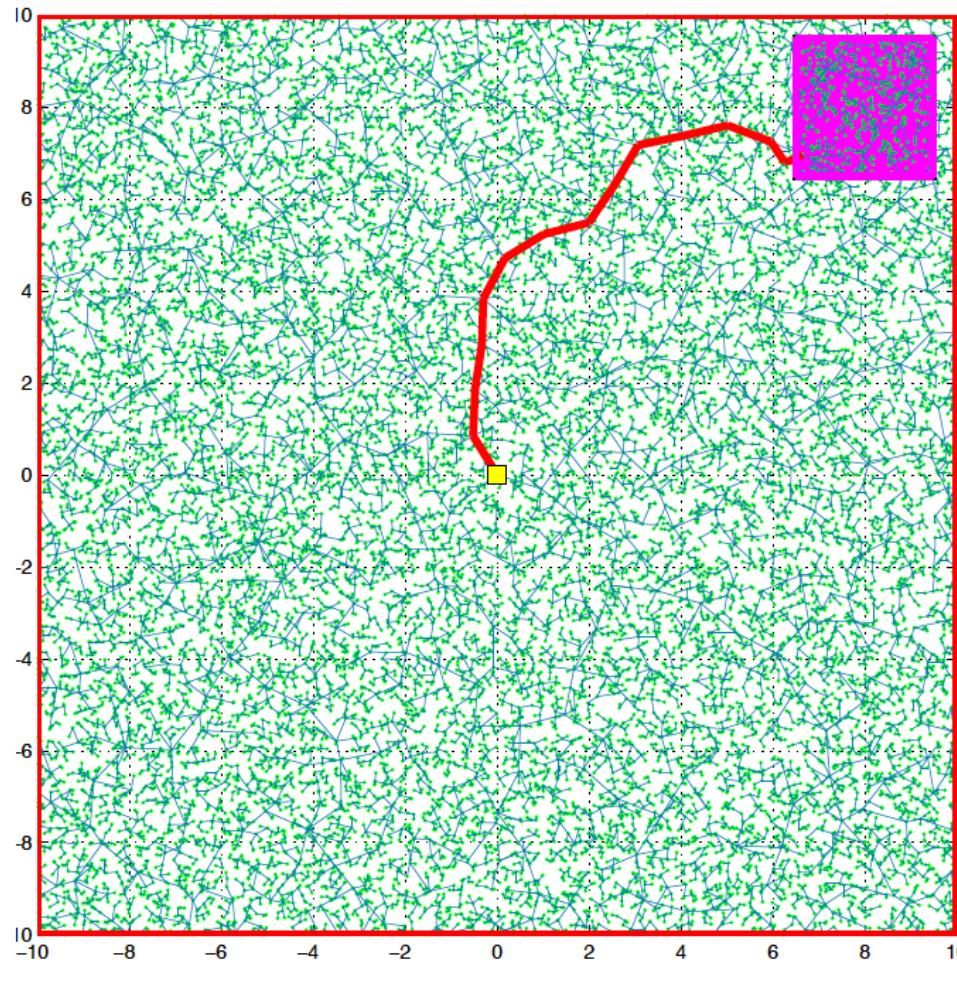
$$C(\mathbf{v}_{new}) + c(\mathbf{v}_{new}, \mathbf{v}) < C(\mathbf{v})$$

i.e., if re-routing through  $\mathbf{v}_{new}$  reduces the path length.

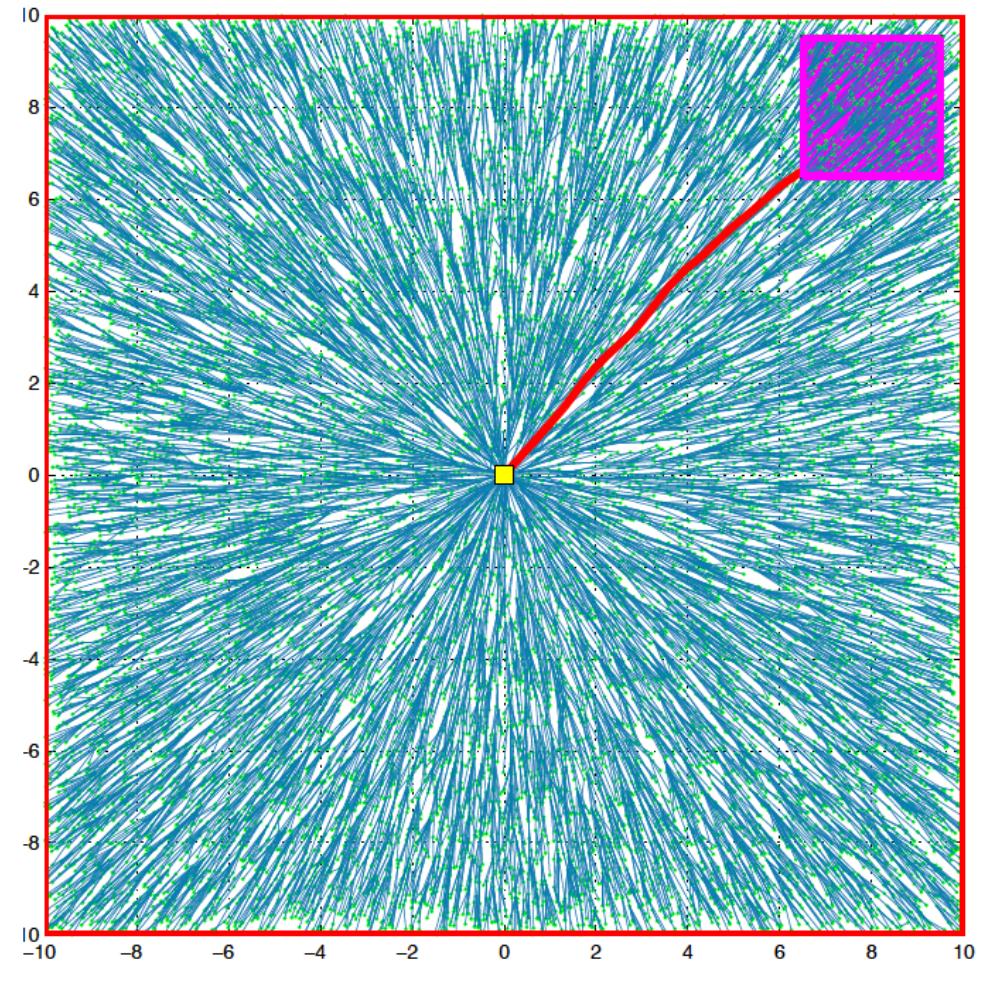
- If so, remove edge between  $\mathbf{v}$  and its parent, and add edge between  $\mathbf{v}$  and  $\mathbf{v}_{new}$ .



# RRT vs. RRT\*



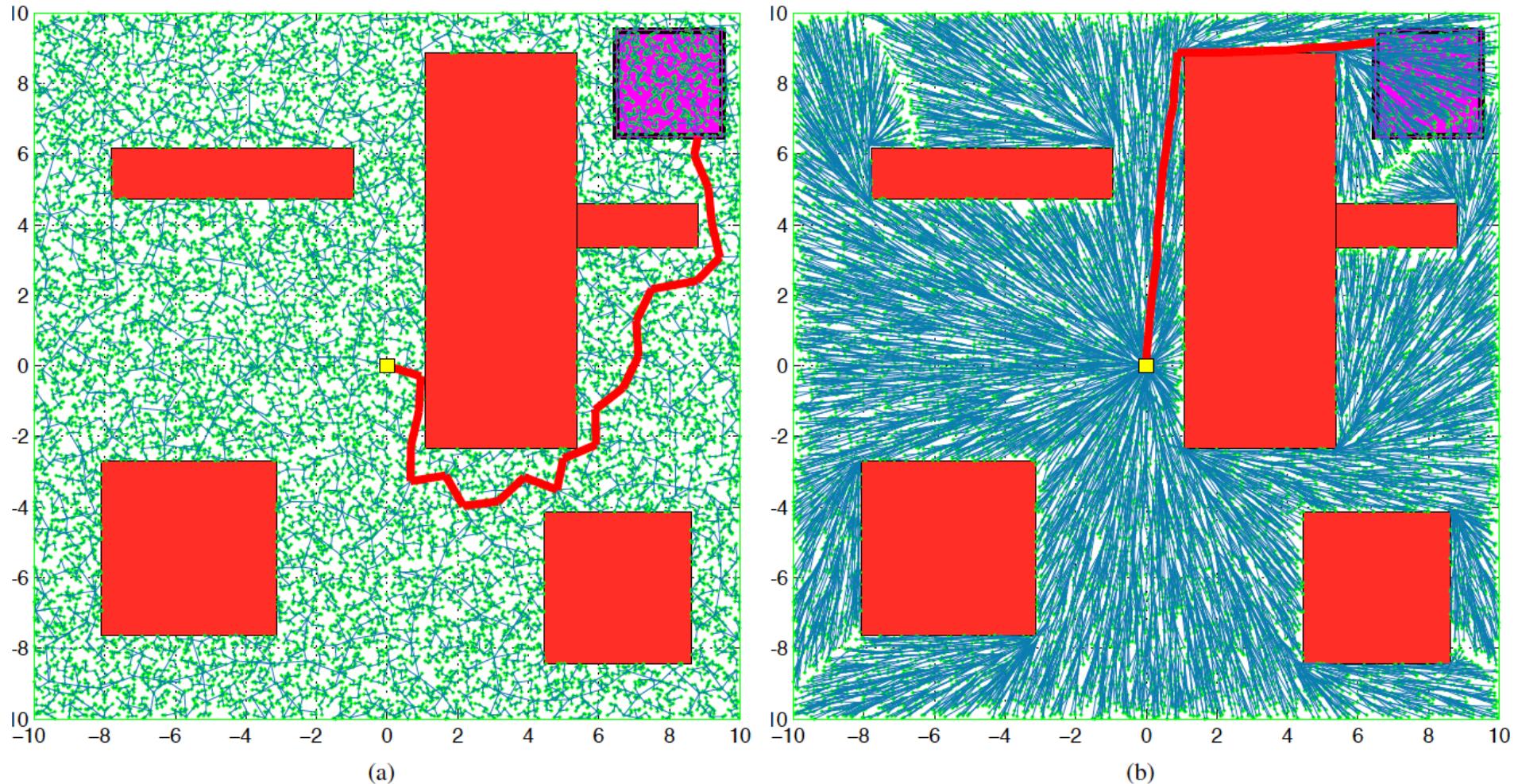
(i)



(j)

From S. Karaman and E. Frazzoli, “Incremental Sampling-based Algorithms for Optimal Motion Planning,” International Journal of Robotic Research, 2010.

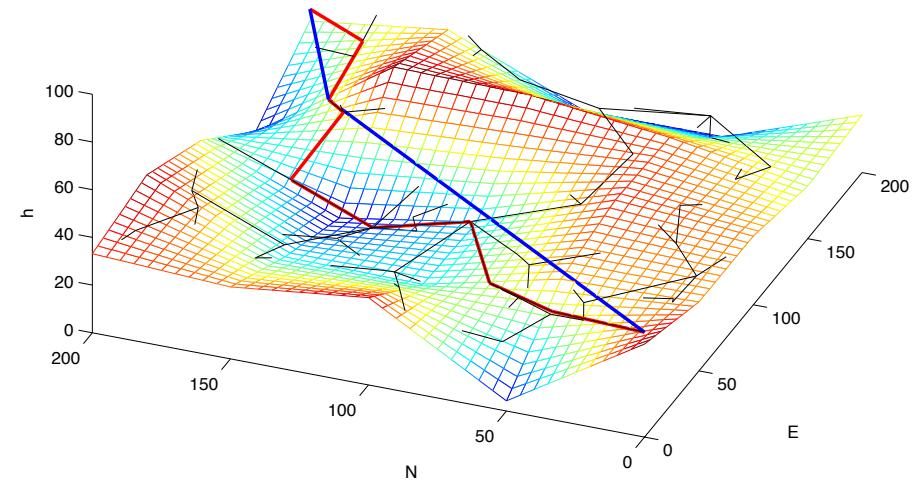
# RRT vs. RRT\*



From S. Karaman and E. Frazzoli, "Incremental Sampling-based Algorithms for Optimal Motion Planning," International Journal of Robotic Research, 2010.

# RRT Path Planning Over 3D Terrain

- Assume terrain map can be queried to determine altitude of terrain at any north-east location
- Must be able to determine altitude for random configuration  $p$  in RRT algorithm
- Must be able to detect collisions with terrain – reject random candidate paths leading to collision
- Options:
  - random altitude within predetermined range
  - random selection of discrete altitudes in desired range
  - set altitude above ground level
- Test candidate paths to ensure flight path angles are feasible – reject if infeasible



# RRT Algorithm – 3D Terrain

---

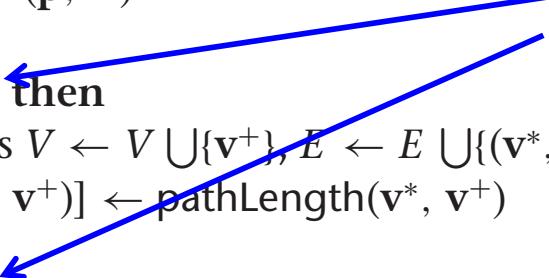
**Algorithm 10** Plan RRT Path:  $\mathcal{W} = \text{planRRT}(\mathcal{T}, \mathbf{p}_s, \mathbf{p}_e)$

---

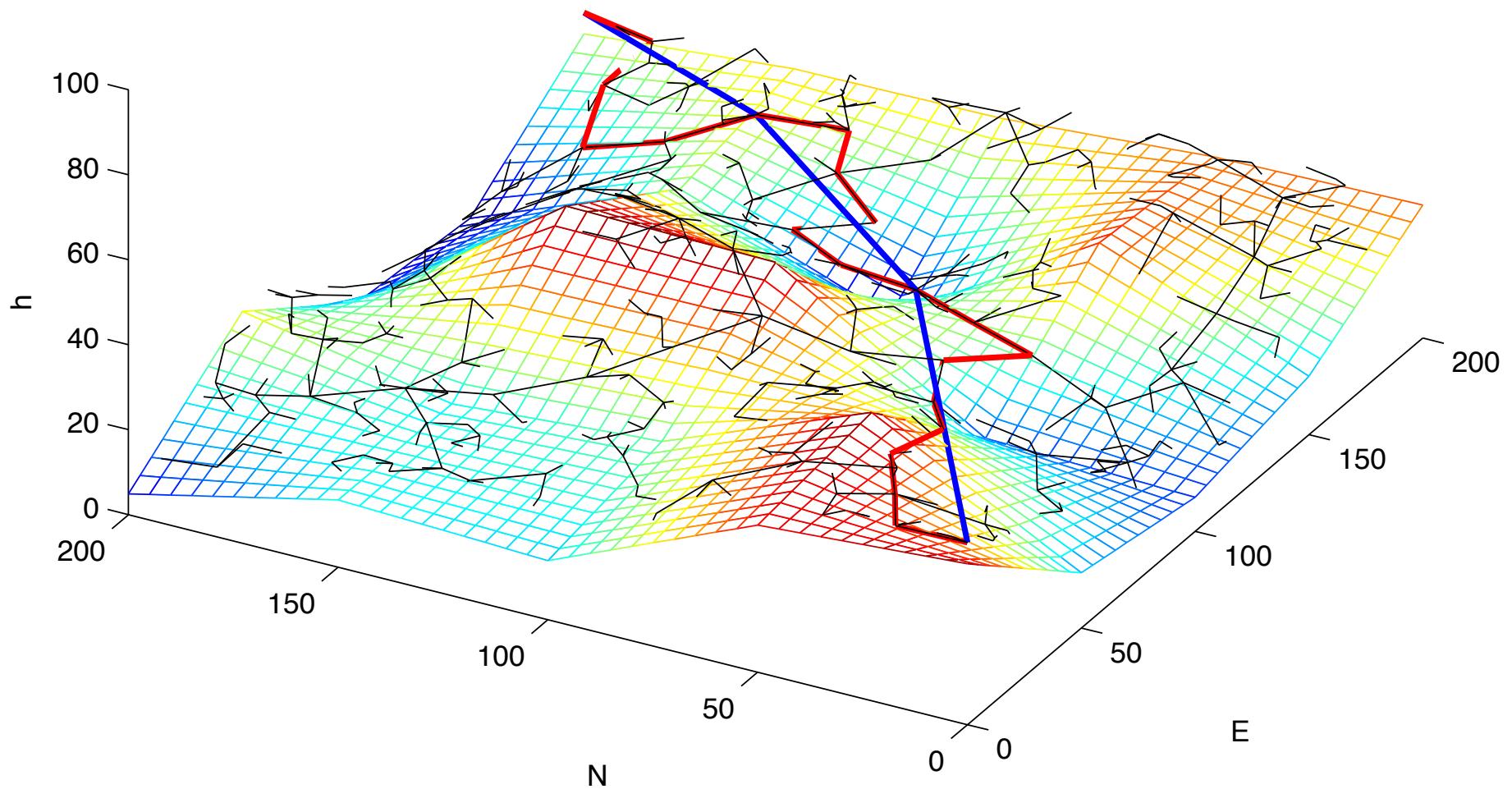
**Input:** Terrain map  $\mathcal{T}$ , start configuration  $\mathbf{p}_s$ , end configuration  $\mathbf{p}_e$

```
1: Initialize RRT graph  $G = (V, E)$  as  $V = \{\mathbf{p}_s\}$ ,  $E = \emptyset$ 
2: while The end node  $\mathbf{p}_e$  is not connected to  $G$ , i.e.,  $\mathbf{p}_e \notin V$  do
3:    $\mathbf{p} \leftarrow \text{generateRandomConfiguration}(\mathcal{T})$ 
4:    $\mathbf{v}^* \leftarrow \text{findClosestConfiguration}(\mathbf{p}, V)$ 
5:    $\mathbf{v}^+ \leftarrow \text{planPath}(\mathbf{v}^*, \mathbf{p}, D)$ 
6:   if  $\text{existFeasiblePath}(\mathcal{T}, \mathbf{v}^*, \mathbf{v}^+)$  then
7:     Update graph  $G = (V, E)$  as  $V \leftarrow V \cup \{\mathbf{v}^+\}$ ,  $E \leftarrow E \cup \{(\mathbf{v}^*, \mathbf{v}^+)\}$ 
8:     Update edge costs as  $C[(\mathbf{v}^*, \mathbf{v}^+)] \leftarrow \text{pathLength}(\mathbf{v}^*, \mathbf{v}^+)$ 
9:   end if
10:  if  $\text{existFeasiblePath}(\mathcal{T}, \mathbf{v}^+, \mathbf{p}_e)$  then
11:    Update graph  $G = (V, E)$  as  $V \leftarrow V \cup \{\mathbf{p}_e\}$ ,  $E \leftarrow E \cup \{(\mathbf{v}^*, \mathbf{p}_e)\}$ 
12:    Update edge costs as  $C[(\mathbf{v}^*, \mathbf{p}_e)] \leftarrow \text{pathLength}(\mathbf{v}^*, \mathbf{p}_e)$ 
13:  end if
14: end while
15:  $\mathcal{W} = \text{findShortestPath}(G, C).$ 
16: return  $\mathcal{W}$ 
```

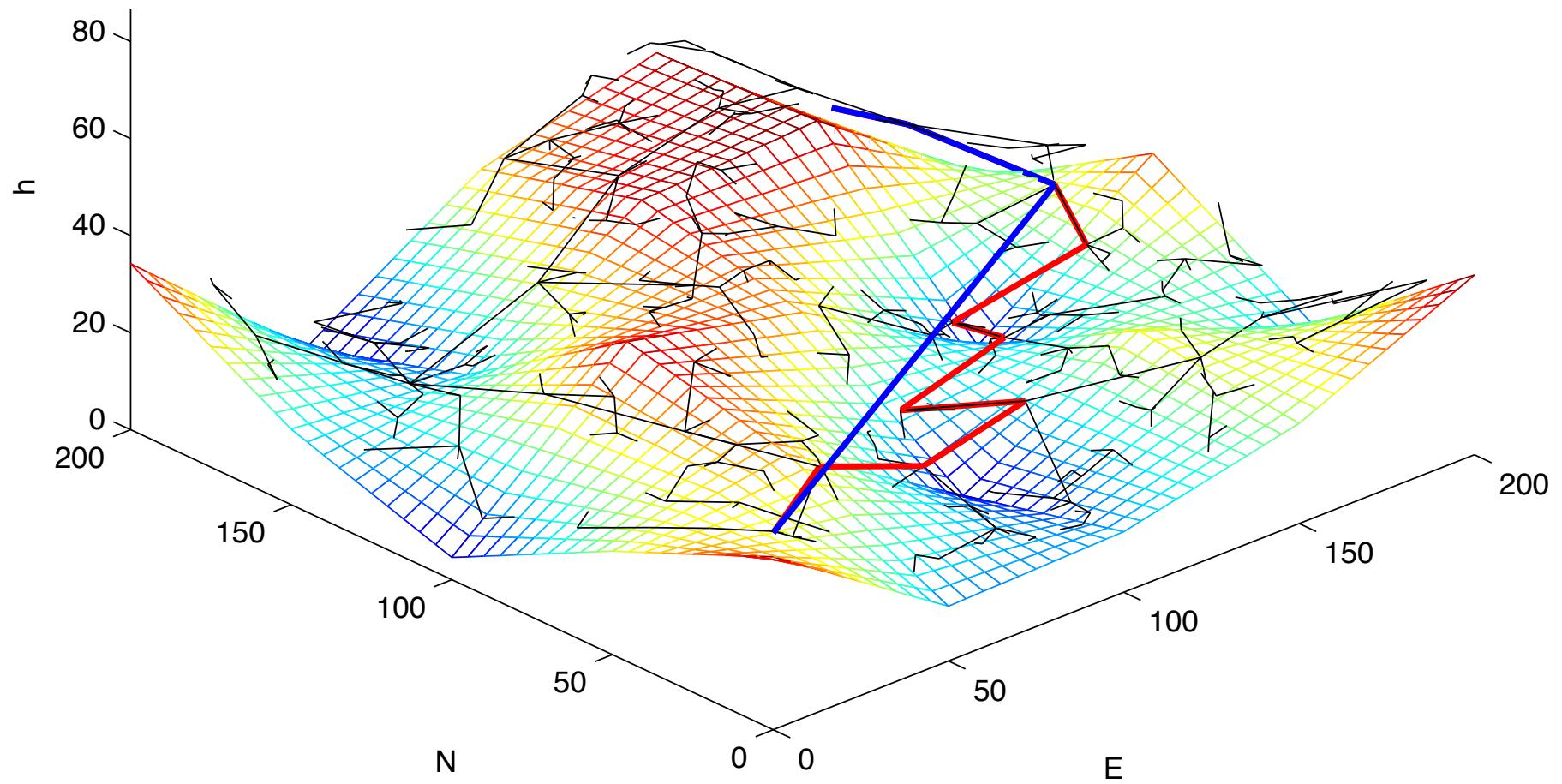
modify to test for collisions  
with terrain and flight path  
angle feasibility



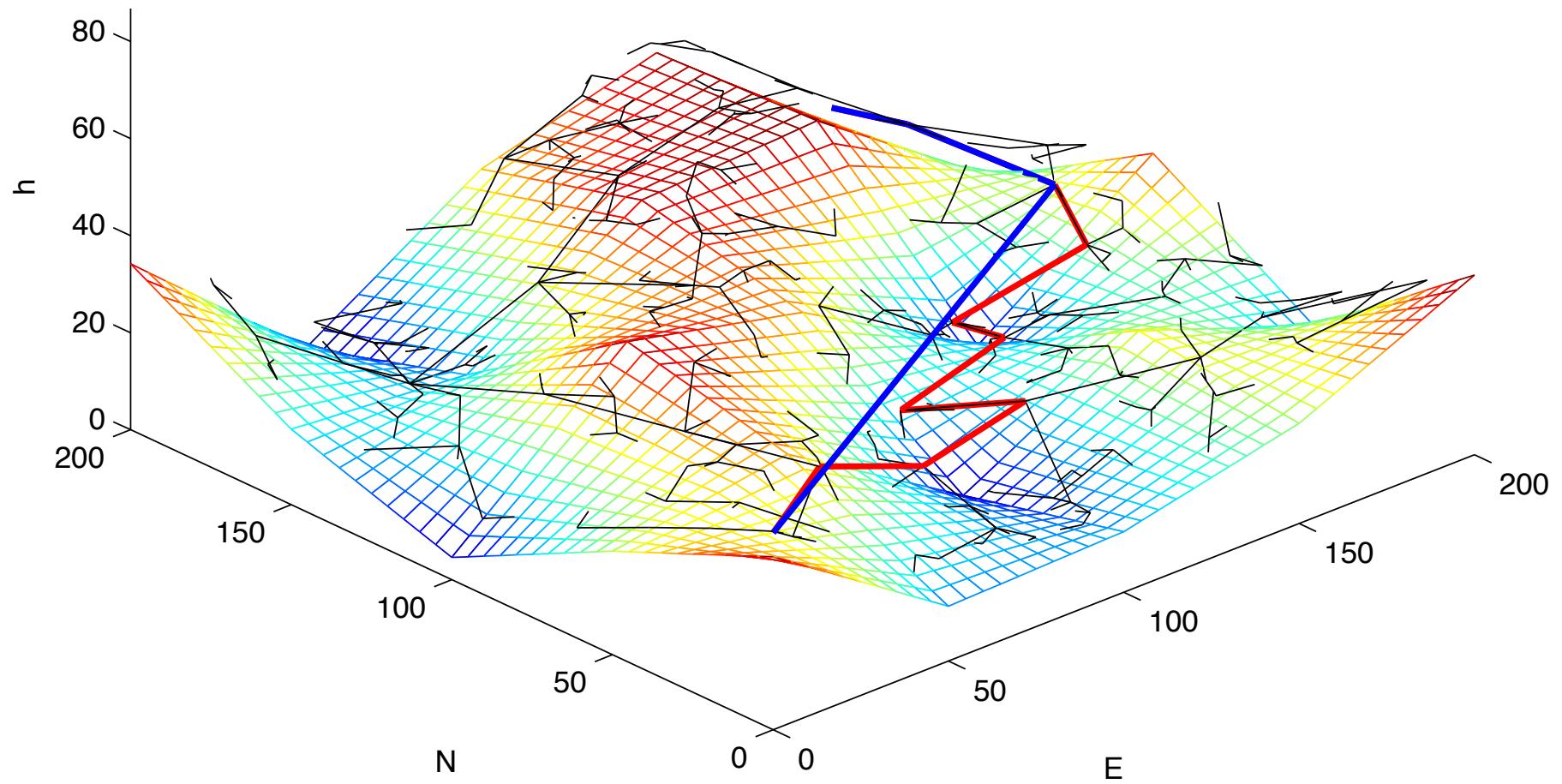
# RRT 3D Terrain Results



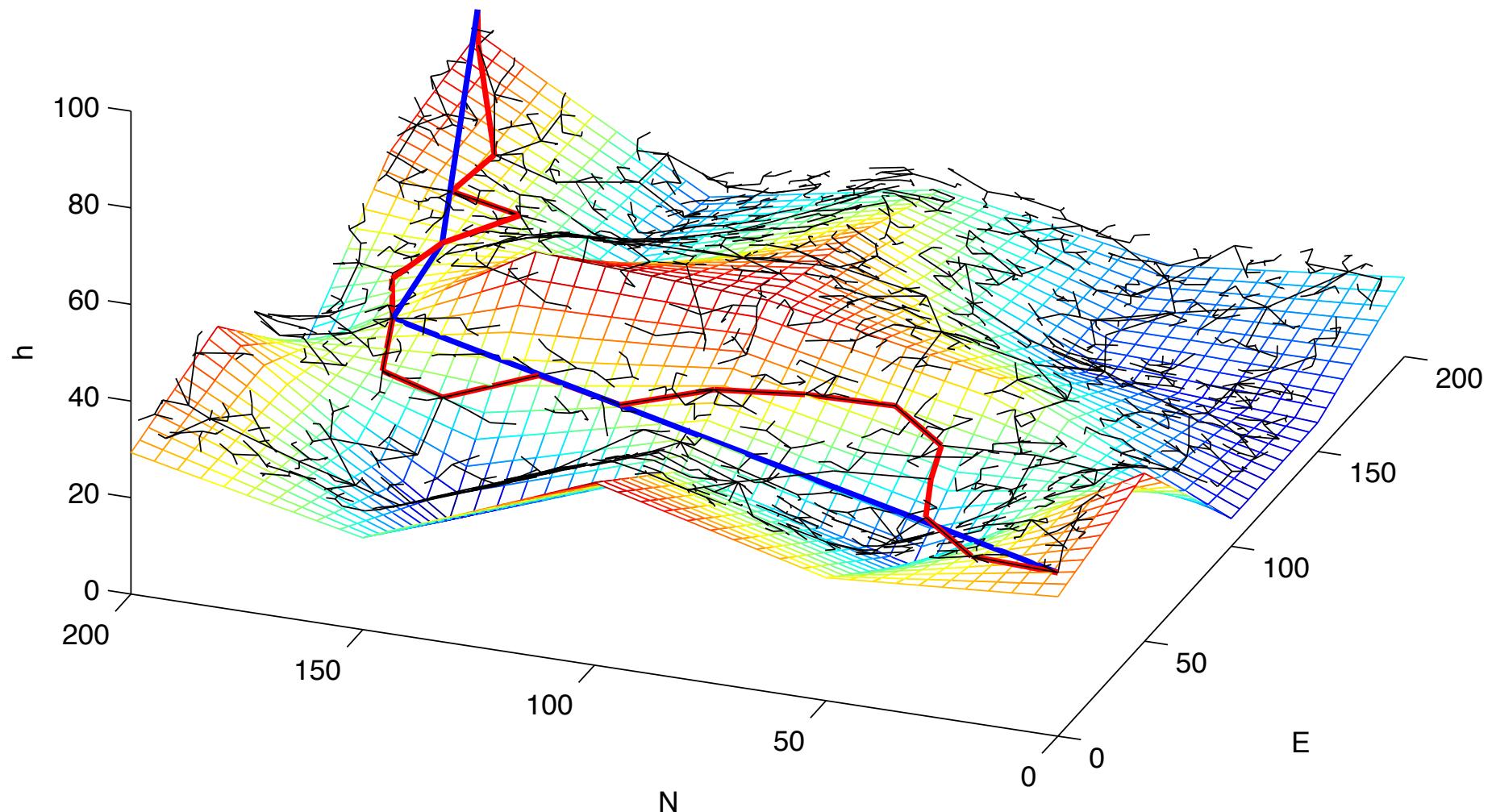
# RRT 3D Terrain Results



# RRT 3D Terrain Results



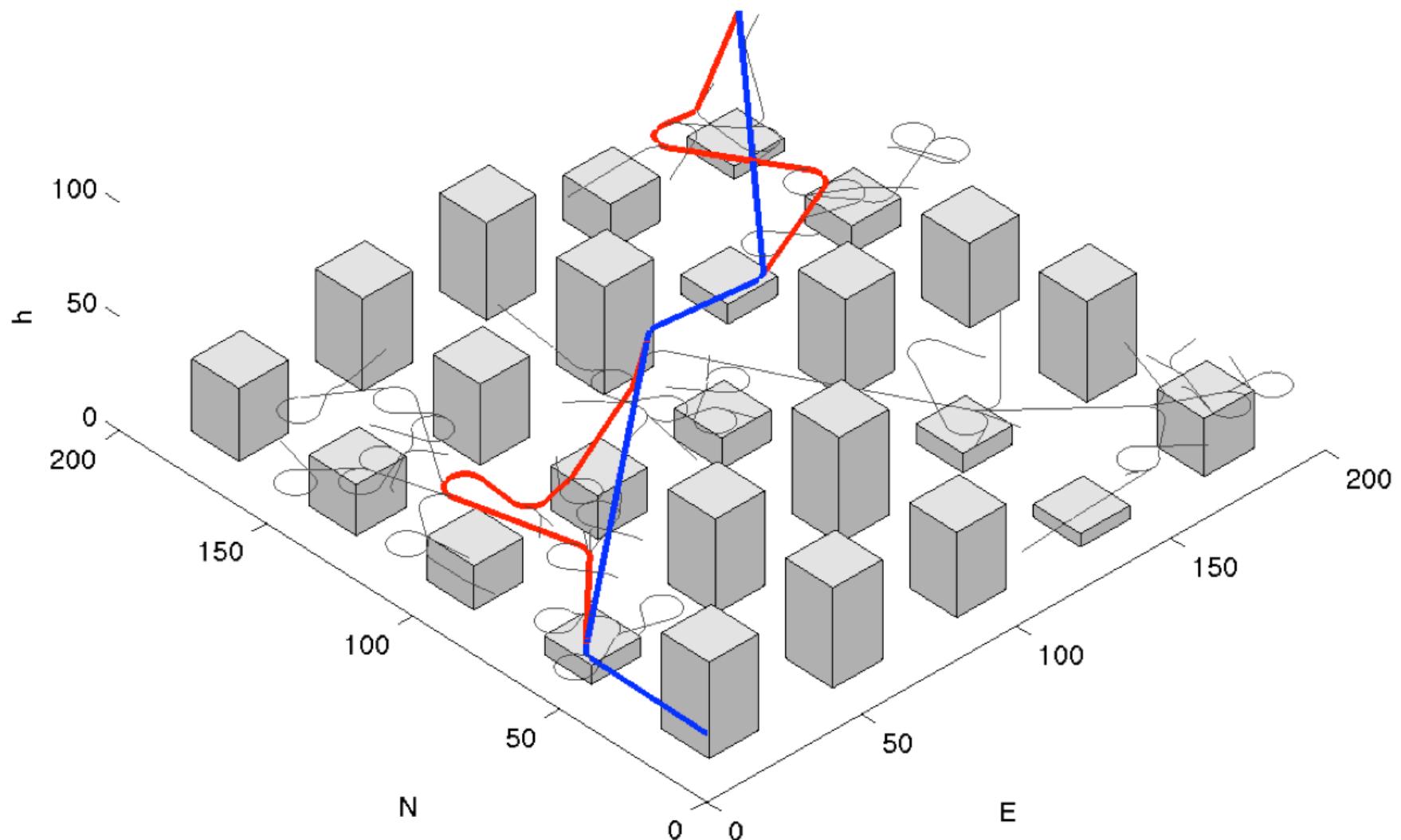
# RRT 3D Terrain Results



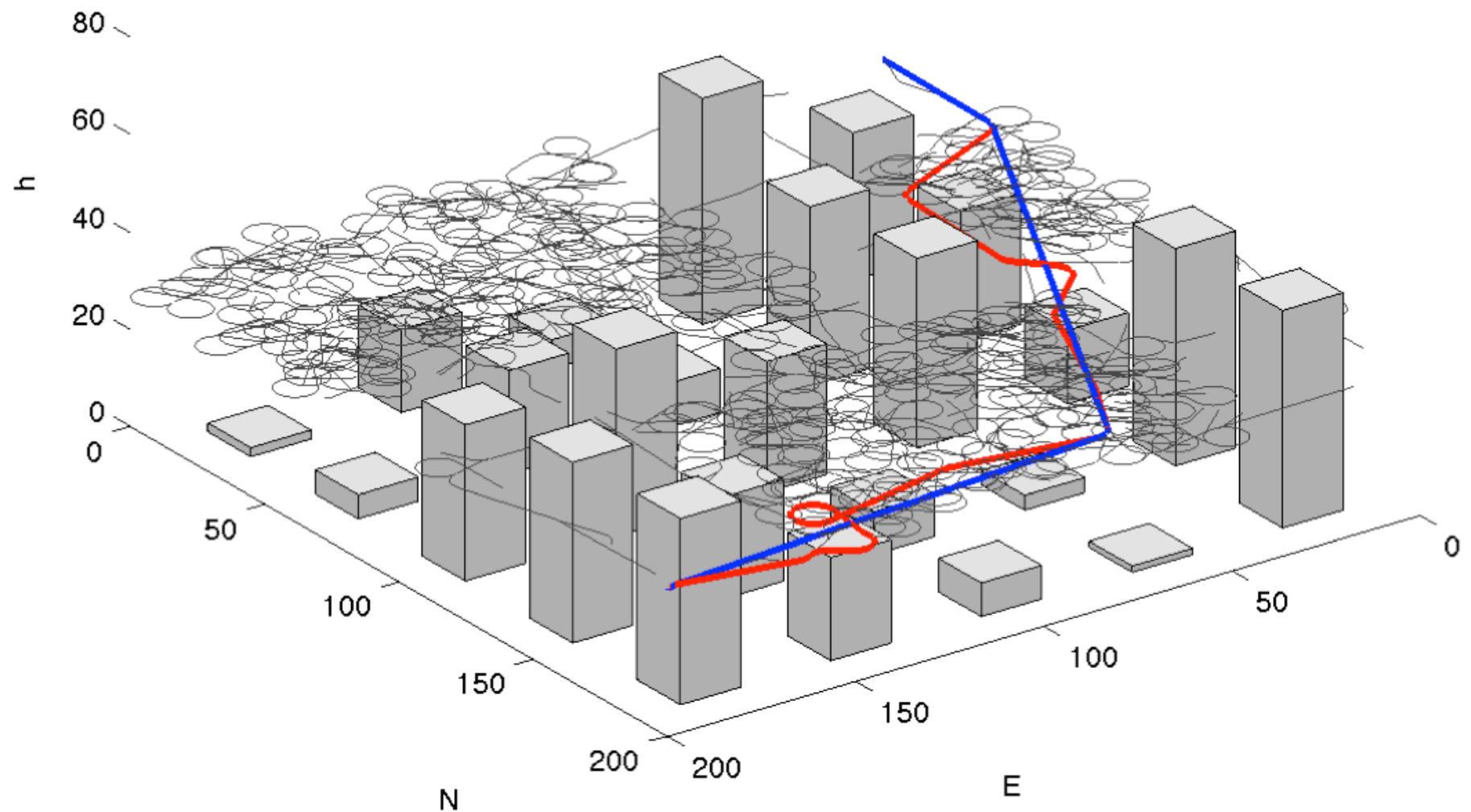
# RRT Dubins Approach

- To generate a new random configuration for tree:
  - Generate random N-E position in environment
  - Find closest node in RRT graph to new random point
  - Select a position of distance  $L$  from the closest RRT node in direction of new point – use this position as N-E coordinates of new configuration
  - Define course angle for new configuration as the angle of the line connecting the RRT graph to the new configuration

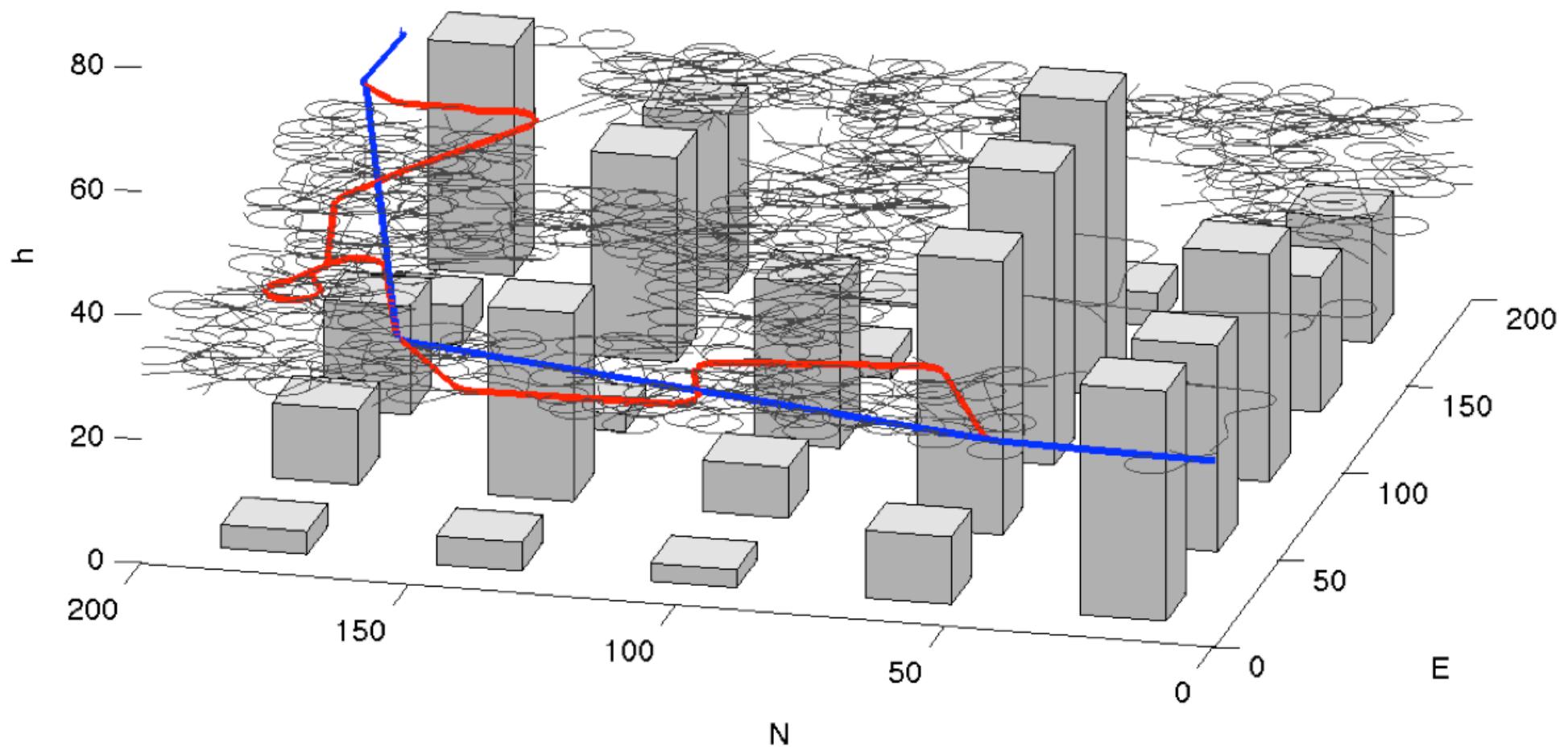
# RRT Dubins Results



# RRT Dubins Results

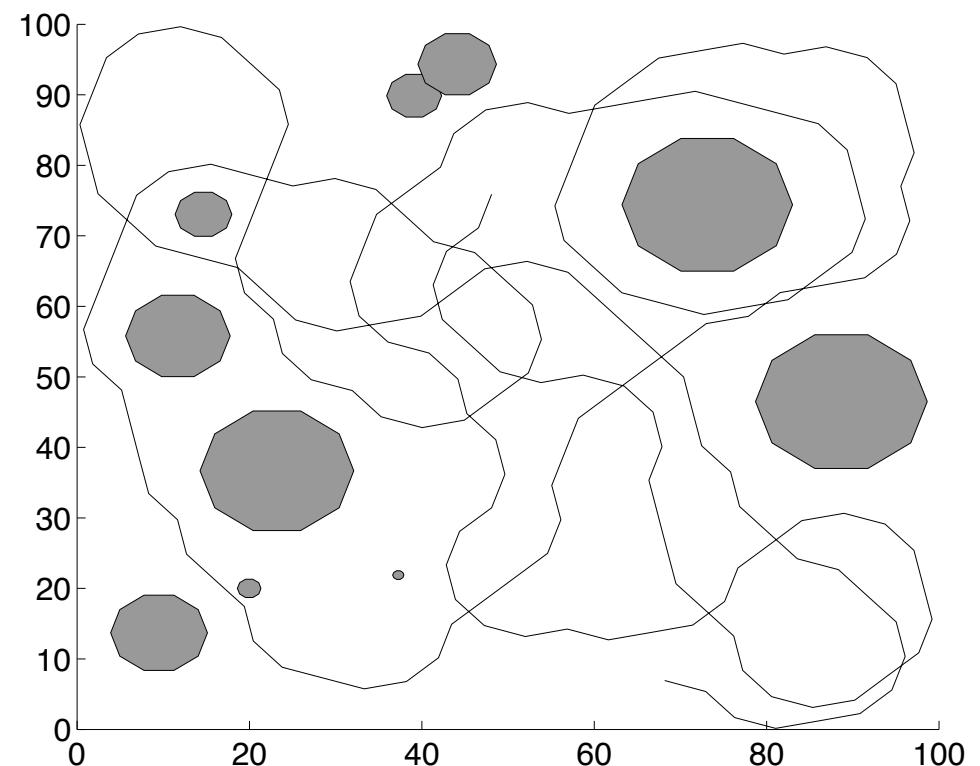


# RRT Dubins Results



# Coverage Algorithms

- Goal: Survey an area
  - Pass sensor footprint over entire area
- Algorithms often cell based
  - Goal: visit every cell



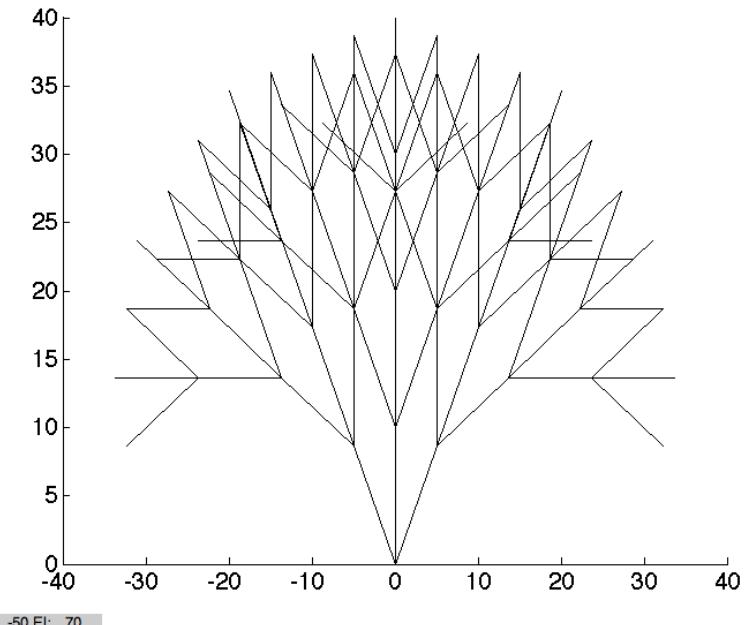
# Coverage Algorithm

- Two maps in memory
  - terrain map
    - used to detect collisions with environment
  - coverage or return map
    - used to track coverage of terrain
- Return map stores value of returning to particular location
  - Return map initialized so that all locations have same return value
  - As locations are visited, return value of that location is decremented by fixed amount:

$$\Upsilon_i[k] = \Upsilon_i[k - 1] - c$$

# Coverage Algorithm

- Finite look ahead tree search used to determine where to go
- Tree generated from current MAV configuration
- Tree searched to determine path that maximizes return value
- Two methods for look ahead tree
  - Uniform branching
    - Predetermined depth
    - Uniform branch length
    - Uniform branch separation
  - RRT



# Coverage Algorithm

---

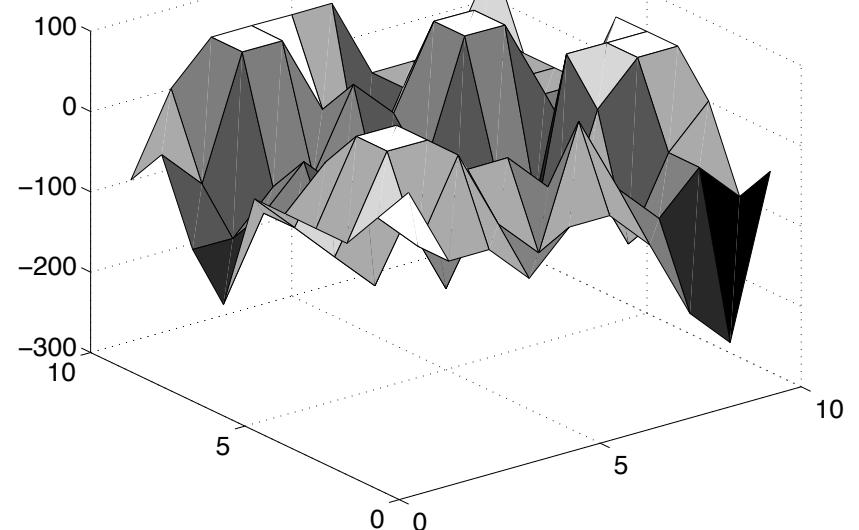
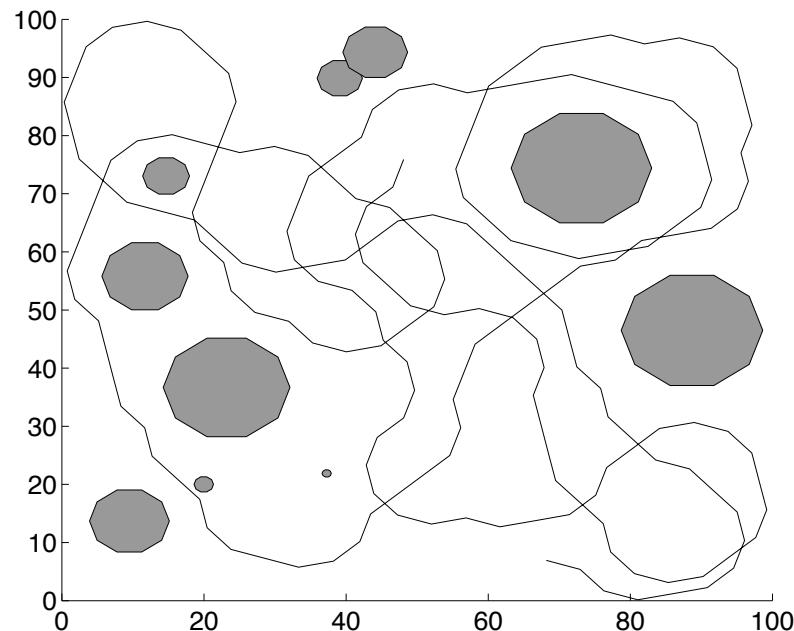
**Algorithm 12** Plan Cover Path:  $\text{planCover}(\mathcal{T}, \Upsilon, p)$ 

---

**Input:** Terrain map  $\mathcal{T}$ , return map  $\Upsilon$ , initial configuration  $p_s$

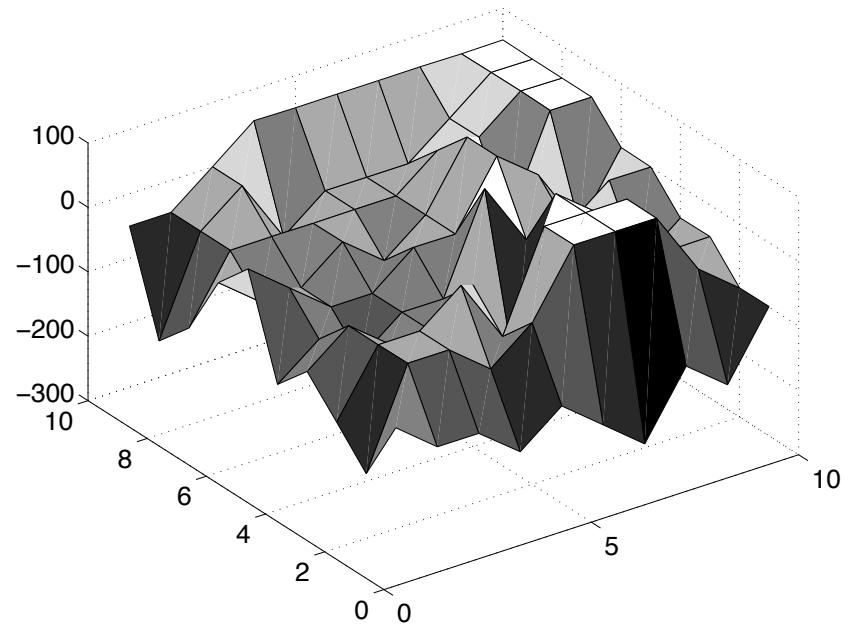
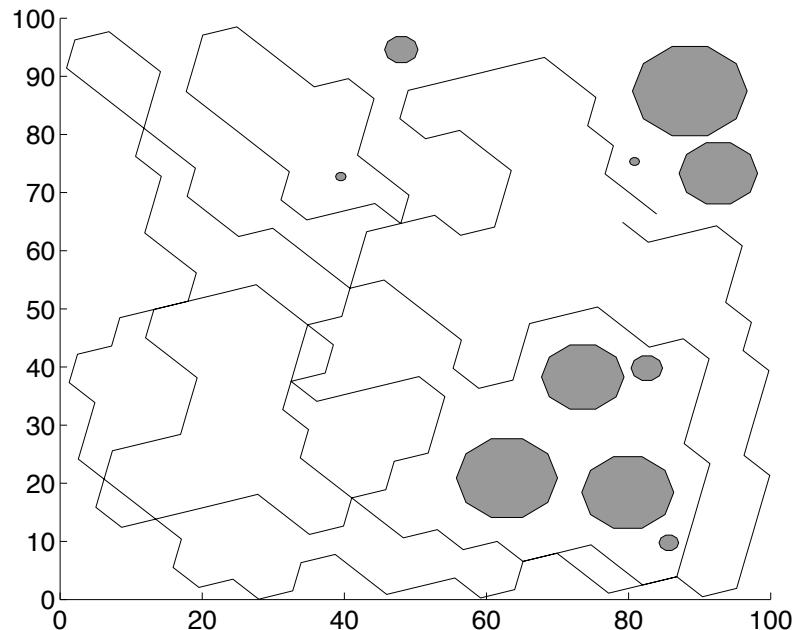
- 1: Initialize look-ahead tree  $G = (V, E)$  as  $V = \{p_s\}$ ,  $E = \emptyset$
  - 2: Initialize return map  $\Upsilon = \{\Upsilon_i : i \text{ indexes the terrain}\}$
  - 3:  $p = p_s$
  - 4: **for** Each planning cycle **do**
  - 5:    $G = \text{generateTree}(p, \mathcal{T}, \Upsilon)$
  - 6:    $\mathcal{W} = \text{highestReturnPath}(G)$
  - 7:   Update  $p$  by moving along the first segment of  $\mathcal{W}$
  - 8:   Reset  $G = (V, E)$  as  $V = \{p\}$ ,  $E = \emptyset$
  - 9:    $\Upsilon = \text{updateReturnMap}(\Upsilon, p)$
  - 10: **end for**
-

# Coverage Planning Results – Uniform Tree



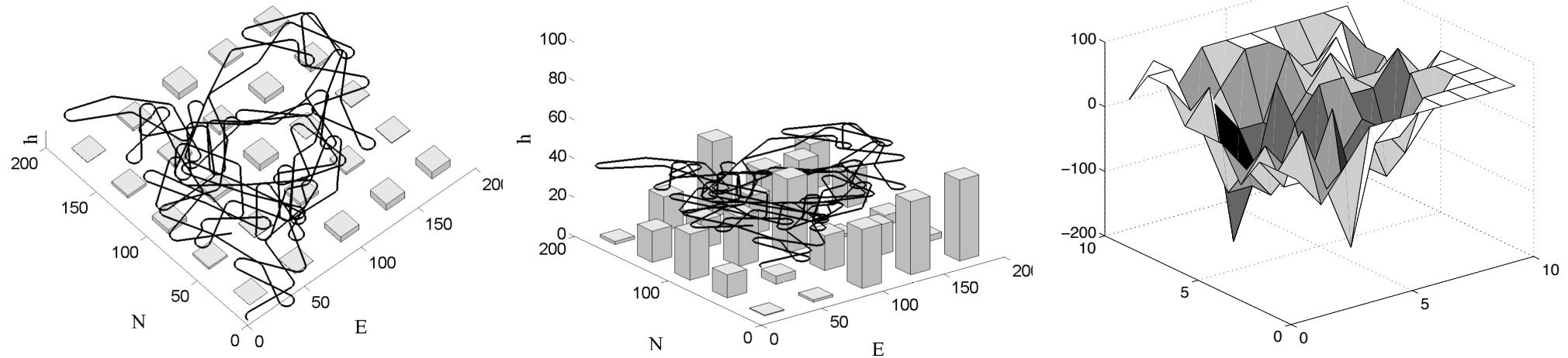
Look ahead length = 5  
Heading change = 30 deg  
Tree depth = 3  
Iterations = 200

# Coverage Planning Results – Uniform Tree



Look ahead length = 5  
Heading change = 60 deg  
Tree depth = 3  
Iterations = 200

# Coverage Planning Results – RRT Dubins



# Coverage Planning Results – RRT Dubins

