BadgerWorks

Topic: Graph Search - Dijkstra, A*, Voronoi Graphs

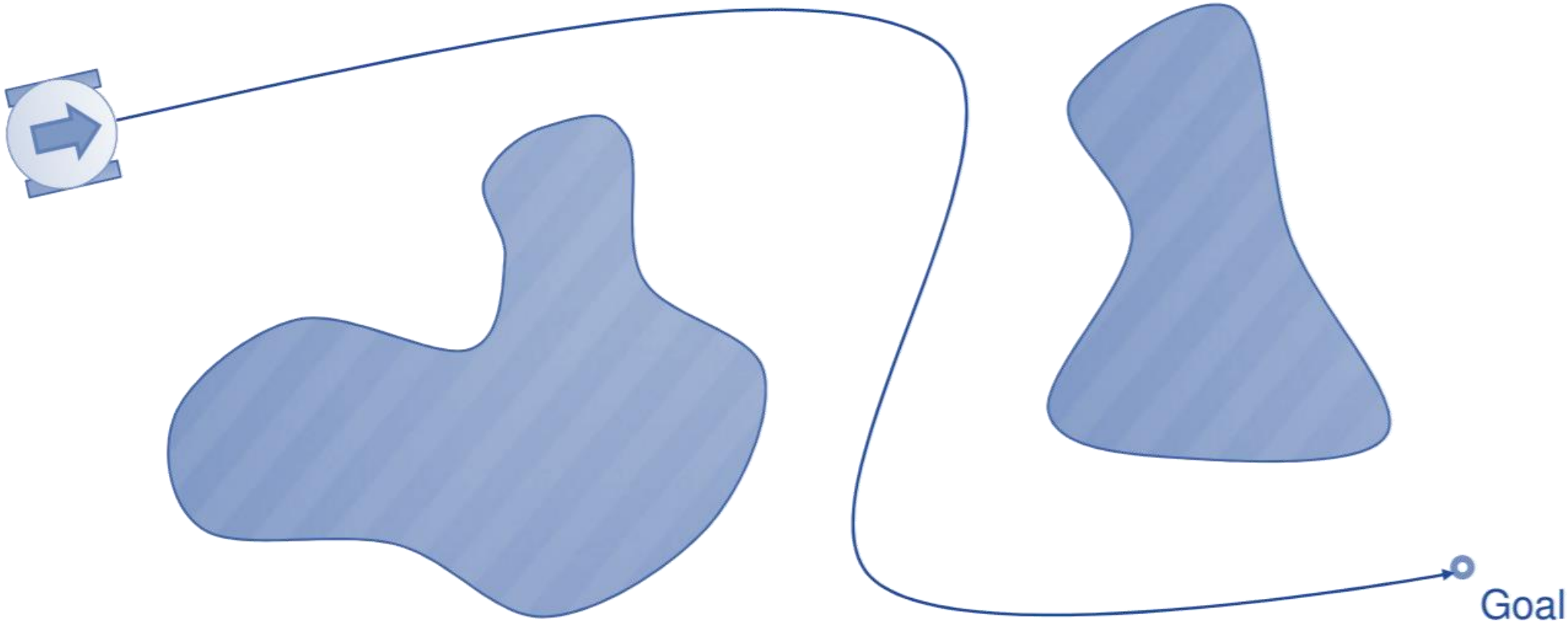
Dr. Kostas Alexis (CSE)

AUTONOMOUS ROBOTS LAB

Graph Search and Voronoi Diagrams

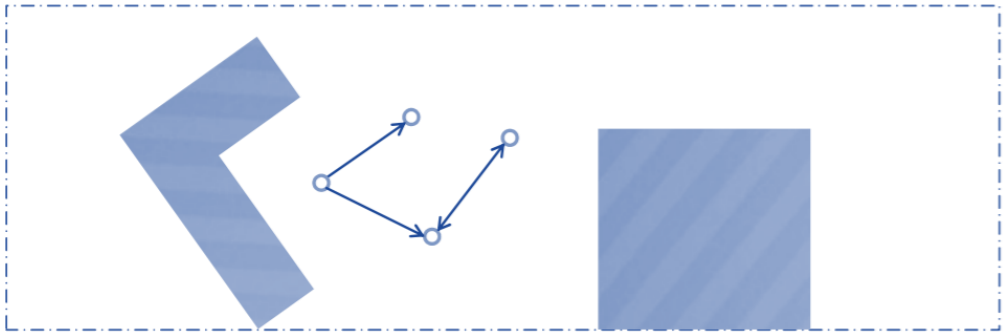
- Goals of this lecture
  - Basic Graph Search algorithms
  - Voronoi diagrams for path planning tasks

The Motion Planning Problem



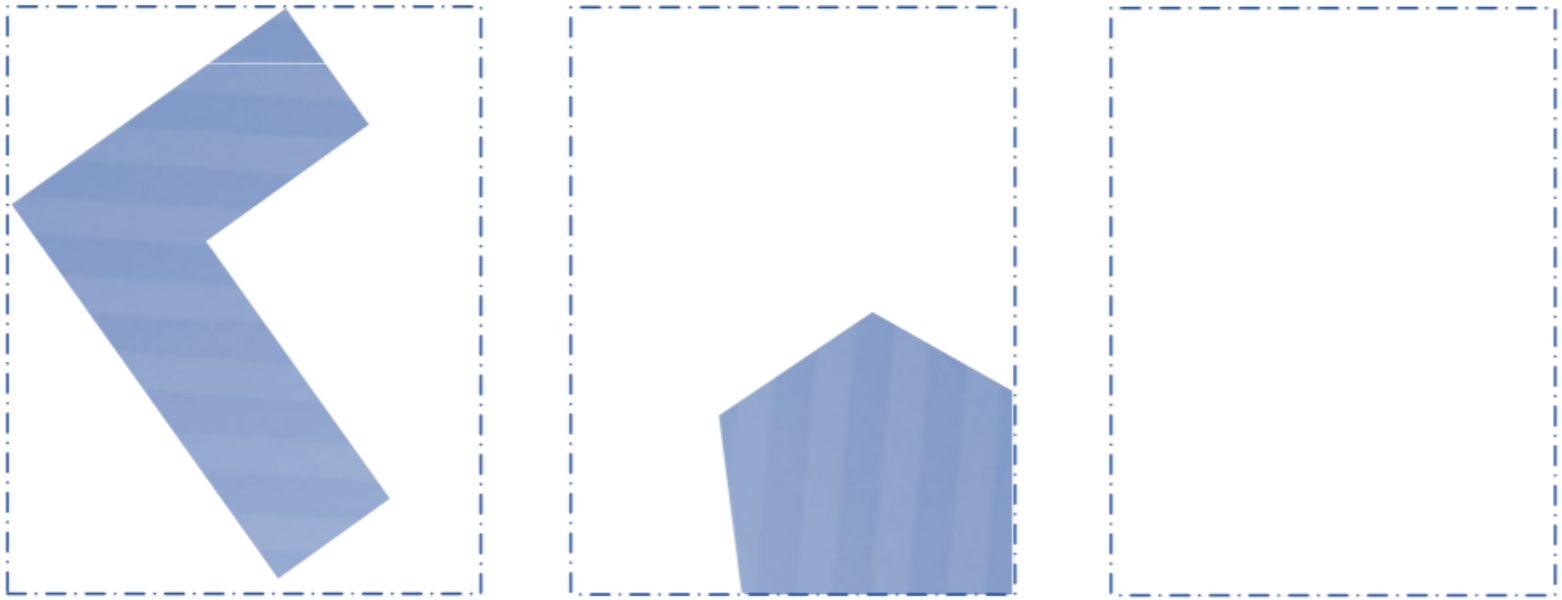
Graph Construction | Overview

- A Graph G (V,E) is characterized by:
  - A set of vertices/nodes V
  - Edges E containing pairs of nodes
- Graphs for motion planning are commonly constructed from map or sensor data.



Graph Construction | Grid and Lattices

- Lattice graphs are largely independent of the workspace representation
- They overlay a repetitive discretization on the workspace



Deterministic Graph Search | Overview

- Encompasses deterministic optimization algorithms operating on graph structures G(V,E)
- The methods find a (globally lowest-cost) connection between a pair of nodes

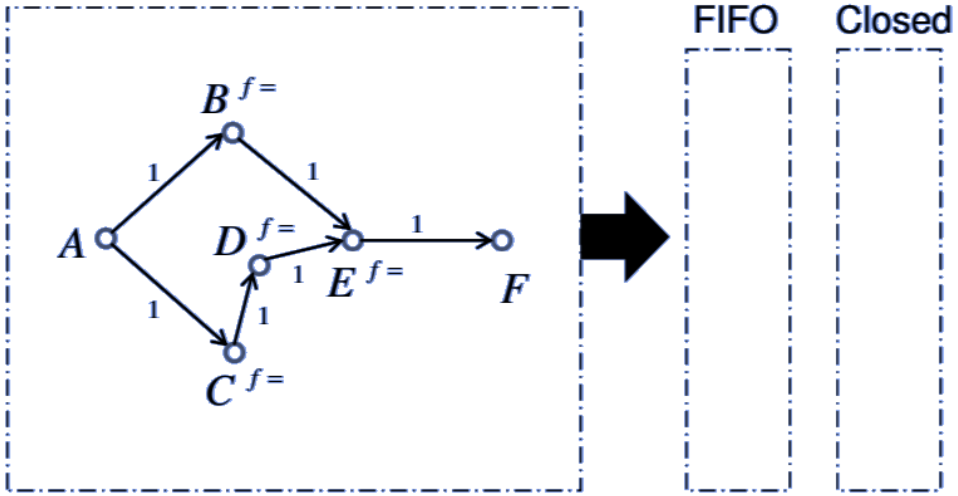
Deterministic Graph Search | Overview

```
Breadth-First-Search(Graph, root):

    create empty set S
    create empty queue Q

    add root to S
    Q.enqueue(root)

    while Q is not empty:
        current = Q.dequeue()
        if current is the goal:
            return current
        for each node n that is adjacent to current:
            if n is not in S:
                add n to S
                Q.enqueue(n)
```

Breadth-First Search | Working principle

- The method expands nodes according to a FIFO queue and a Closed list
- It backtracks the solution from the goal state backwards in a greedy way

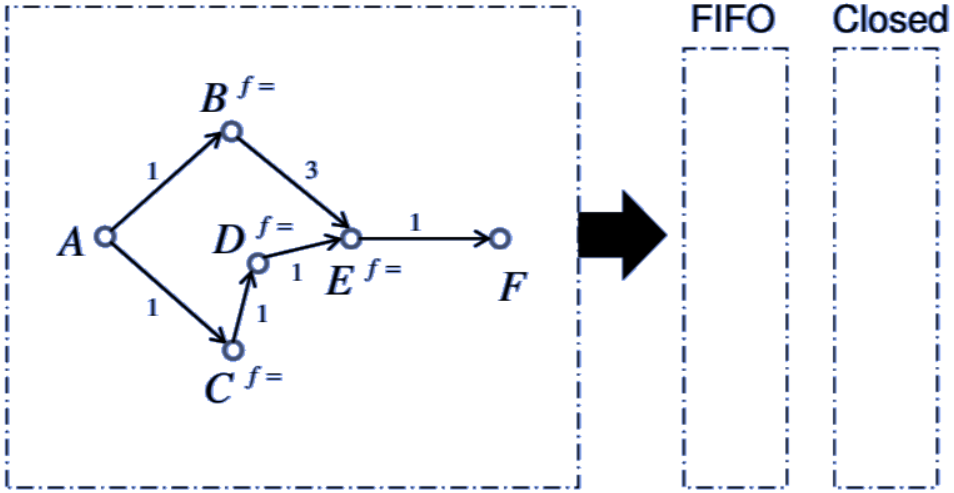
```
BF(Graph G, Node Start, Node Goal)
        Queue.init(FIFO)
        Queue.push(Start)
        while Queue is not empty
                Node curr = Queue.pop()
                if curr is Goal return
                Closed.push(curr)
                Nodes next = expand(curr)
                for all next not in Closed:
                        Queue.push(next)
```



Breadth-First Search | Working principle

- The method expands nodes according to a FIFO queue and a Closed list
- It backtracks the solution from the goal state backwards in a greedy way

```
BF(Graph G, Node Start, Node Goal)
        Queue.init(FIFO)
        Queue.push(Start)
        while Queue is not empty
                Node curr = Queue.pop()
                if curr is Goal return
                Closed.push(curr)
                Nodes next = expand(curr)
                for all next not in Closed:
                        Queue.push(next)
```



Breadth-First Search | Properties

- The trajectory to the first goal state encountered is optimal if all edge costs on the graph are identical and positive
- Optimality of the solution is retained for arbitrary positive edge costs, if search is continued until queue is empty
- Breadth-first search has a time complexity of O (|V| + |E|)

Dijkstra's Search | Working principle

- Dijkstra's search expands nodes according to a HEAP and a Closed list
- It backtracks the solution from the goal state backwards in a greedy way

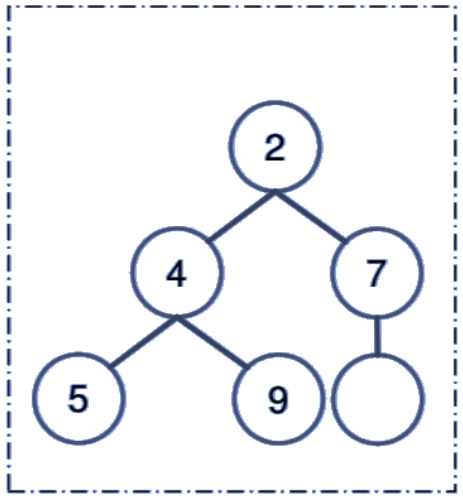
Dijkstra's Search | Working principle

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:             // Initialization
        dist[v] ← INFINITY                  // Unknown distance from source to v
        prev[v] ← UNDEFINED                 // Previous node in optimal path from source
        add v to Q                          // All nodes initially in Q (unvisited nodes)

    dist[source] ← 0                        // Distance from source to source

    while Q is not empty:
        u ← vertex in Q with min dist[u]    // Node with the least distance will be selected first
        remove u from Q

        for each neighbor v of u:           // where v is still in Q.
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:               // A shorter path to v has been found
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

Dijkstra's Search | Working principle

- Dijkstra's search expands nodes according to a HEAP and a Closed list
- It backtracks the solution from the goal state backwards in a greedy way

```
Min_Bin_Heap_Push(Node up)
        insert up at end of heap
                while up < parent(up):
                        swap(up,parent(up))
```

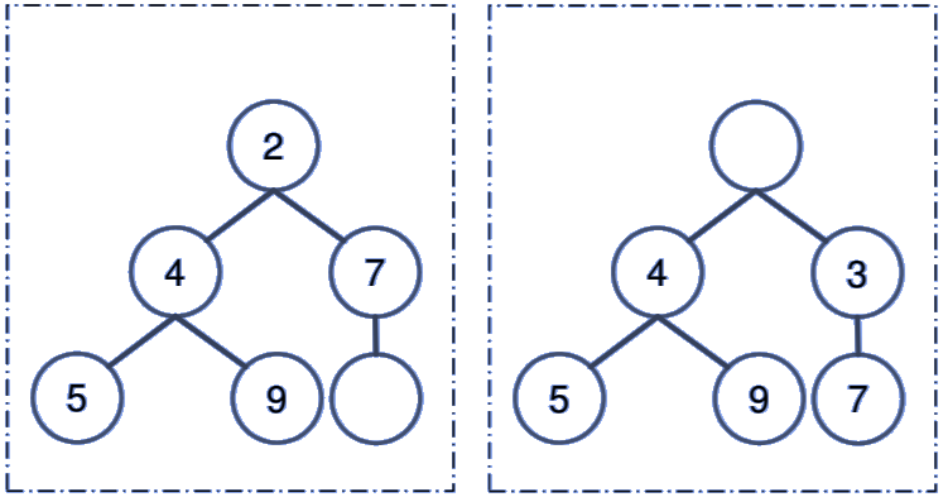


Dijkstra's Search | Working principle

- Dijkstra's search expands nodes according to a HEAP and a Closed list
- It backtracks the solution from the goal state backwards in a greedy way

```
Min_Bin_Heap_Push(Node up)
        insert up at end of heap
                while up < parent(up):
                        swap(up,parent(up))
```

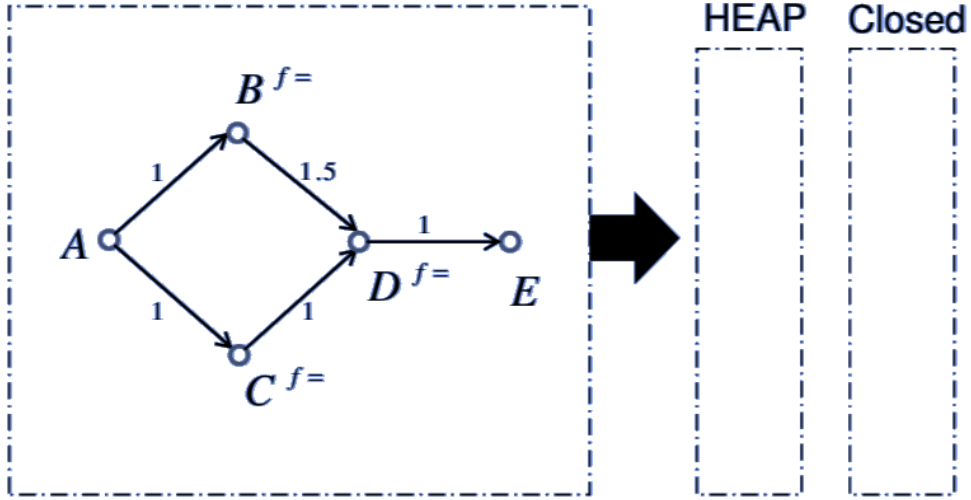
```
Min_Bin_Heap_Pop()
        return top element of heap
        move bottom element to top as down
                while up < parent(up):
                        swap(up,parent(up))
```



Dijkstra's Search | Working principle

- Dijkstra's search expands nodes according to a HEAP and a Closed list
- It backtracks the solution from the goal state backwards in a greedy way

```
Dijkstra(Graph G, Node Start, Node End)
        Queue.init(BIN_MIN_HEAP)
        Queue.push(Start)
        while Queue is not empty:
                Node curr = Queue.pop()
                if curr is Goal return
                Closed.puhs(curr)
                Nodes next = expand(curr)
                for all next not in Closed:
                        Queue.push(next)
```



Dijkstra's Search | Properties & Reqs

- The sequence to the first goal state encountered is optimal
- Edge costs must be strictly positive – otherwise, employ Bellman-Ford
- Dijkstra's search has a time complexity of $O(|V|\log|V|+|E|)$

The A* Algorithm | Working principle

- A* expands nodes according to a HEAP and a Closed list
- It makes use of a heuristic function to guide the search
- It backtracks the solution from the goal state backwards in a greedy way

The A* Algorithm | Working principle

```
function A*(start, goal)
    // The set of nodes already evaluated
    closedSet := {}

    // The set of currently discovered nodes that are not evaluated yet.
    // Initially, only the start node is known.
    openSet := {start}

    // For each node, which node it can most efficiently be reached from.
    // If a node can be reached from many nodes, cameFrom will eventually contain the
    // most efficient previous step.
    cameFrom := the empty map

    // For each node, the cost of getting from the start node to that node.
    gScore := map with default value of Infinity

    // The cost of going from start to start is zero.
    gScore[start] := 0

    // For each node, the total cost of getting from the start node to the goal
    // by passing by that node. That value is partly known, partly heuristic.
    fScore := map with default value of Infinity

    // For the first node, that value is completely heuristic.
    fScore[start] := heuristic_cost_estimate(start, goal)

    while openSet is not empty
        current := the node in openSet having the lowest fScore[] value
        if current = goal
            return reconstruct_path(cameFrom, current)

        openSet.Remove(current)
        closedSet.Add(current)

        for each neighbor of current
            if neighbor in closedSet
                continue        // Ignore the neighbor which is already evaluated.

            if neighbor not in openSet  // Discover a new node
                openSet.Add(neighbor)

            // The distance from start to a neighbor
            tentative_gScore := gScore[current] + dist_between(current, neighbor)
            if tentative_gScore >= gScore[neighbor]
                continue        // This is not a better path.

            // This path is the best until now. Record it!
            cameFrom[neighbor] := current
            gScore[neighbor] := tentative_gScore
            fScore[neighbor] := gScore[neighbor] + heuristic_cost_estimate(neighbor, goal)

    return failure
```

```
function reconstruct_path(cameFrom, current)
    total_path := [current]
    while current in cameFrom.Keys:
        current := cameFrom[current]
        total_path.append(current)
    return total_path
```

The A* Algorithm | Working principle

```
function A*(start, goal)
    // The set of nodes already evaluated
    closedSet := {}

    // The set of currently discovered nodes that are not evaluated yet.
    // Initially, only the start node is known.
    openSet := {start}

    // For each node, which node it can most efficiently be reached from.
    // If a node can be reached from many nodes, cameFrom will eventually contain the
    // most efficient previous step.
    cameFrom := the empty map

    // For each node, the cost of getting from the start node to that node.
    gScore := map with default value of Infinity

    // The cost of going from start to start is zero.
    gScore[start] := 0

    // For each node, the total cost of getting from the start node to the goal
    // by passing by that node. That value is partly known, partly heuristic.
    fScore := map with default value of Infinity

    // For the first node, that value is completely heuristic.
    fScore[start] := heuristic_cost_estimate(start, goal)
```

The A* Algorithm | Working principle

```
while openSet is not empty
    current := the node in openSet having the lowest fScore[] value
    if current = goal
        return reconstruct_path(cameFrom, current)

    openSet.Remove(current)
    closedSet.Add(current)

    for each neighbor of current
        if neighbor in closedSet
            continue        // Ignore the neighbor which is already evaluated.

        if neighbor not in openSet  // Discover a new node
            openSet.Add(neighbor)

        // The distance from start to a neighbor
        tentative_gScore := gScore[current] + dist_between(current, neighbor)
        if tentative_gScore >= gScore[neighbor]
            continue        // This is not a better path.

        // This path is the best until now. Record it!
        cameFrom[neighbor] := current
        gScore[neighbor] := tentative_gScore
        fScore[neighbor] := gScore[neighbor] + heuristic_cost_estimate(neighbor, goal)

return failure
```

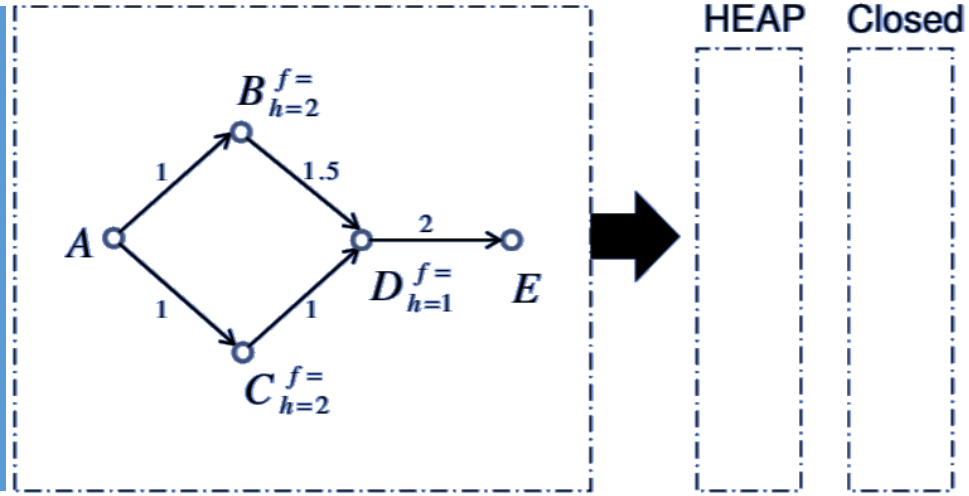
The A* Algorithm | Working principle

```
function reconstruct_path(cameFrom, current)
    total_path := [current]
    while current in cameFrom.Keys:
        current := cameFrom[current]
        total_path.append(current)
    return total_path
```

The A* Algorithm | Working principle

- A* expands nodes according to a HEAP and a Closed list
- It makes use of a heuristic function to guide the search
- It backtracks the solution from the goal state backwards in a greedy way

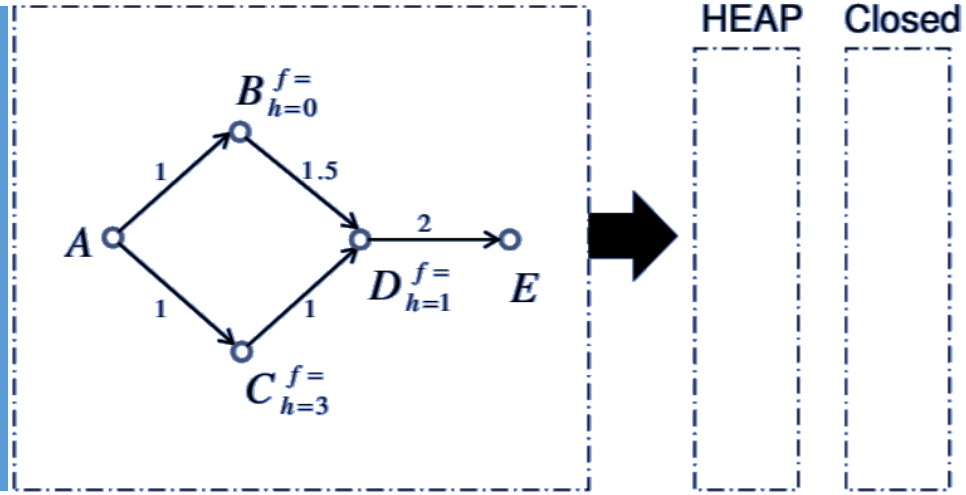
```
A_star(Graph G, Heap H, Node Start, Node Goal)
        Queue.init(BIN_MIN_HEAP,H)
        Queue.push(Start)
        while Queue is not empty:
                Node curr = Queue.pop()
                if curr is Goal return
                Closed.push(curr)
                Nodes next = expand(curr)
                for all next not in Closed:
                        Queue.push(next)
```



The A* Algorithm | Working principle

- A* expands nodes according to a HEAP and a Closed list
- It makes use of a heuristic function to guide the search
- It backtracks the solution from the goal state backwards in a greedy way

```
A_star(Graph G, Heap H, Node Start, Node Goal)
        Queue.init(BIN_MIN_HEAP,H)
        Queue.push(Start)
        while Queue is not empty:
                Node curr = Queue.pop()
                if curr is Goal return
                Closed.push(curr)
                Nodes next = expand(curr)
                for all next not in Closed:
                        Queue.push(next)
```

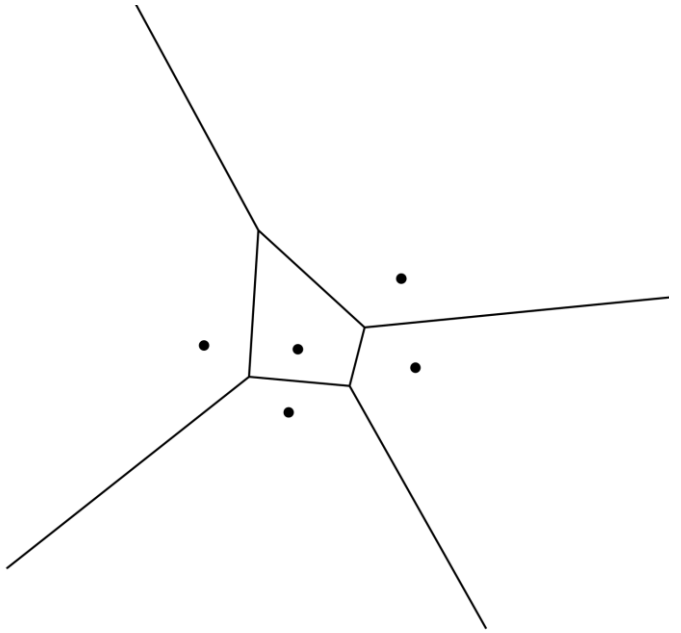


The A* Algorithm | Properties & Reqs

- The trajectory to the first goal state encountered is optimal
- Edge costs must be strictly positive
- For optimality to hold heuristic must be consistent

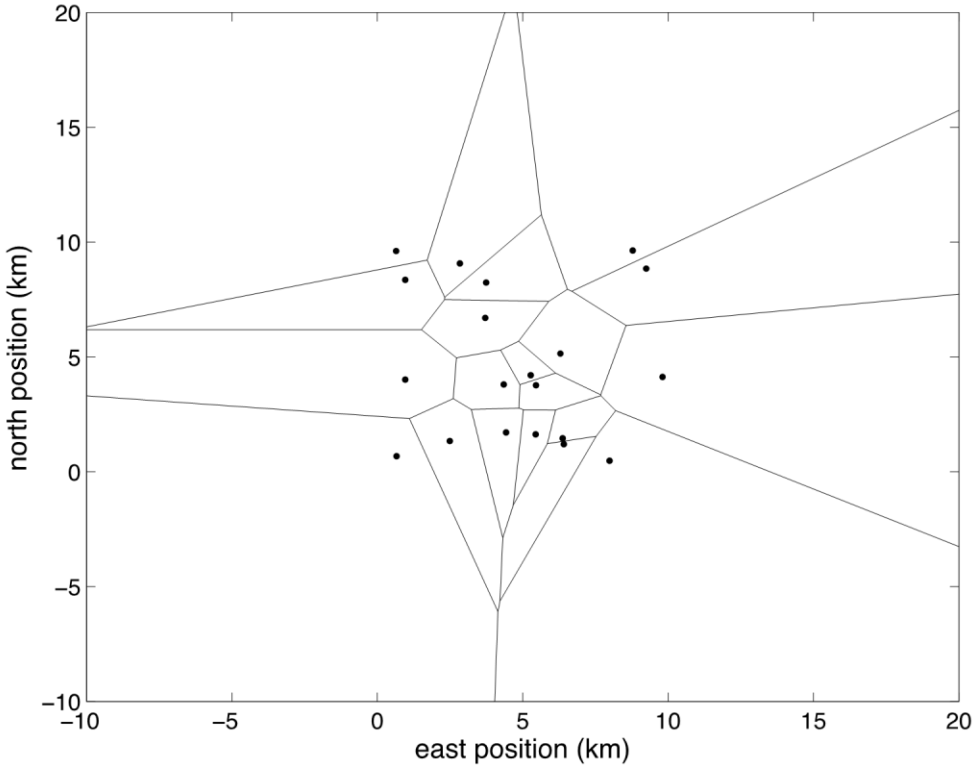
Voronoi Graphs

- For finite number Q of point obstacles, Voronoi graph divides search space 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



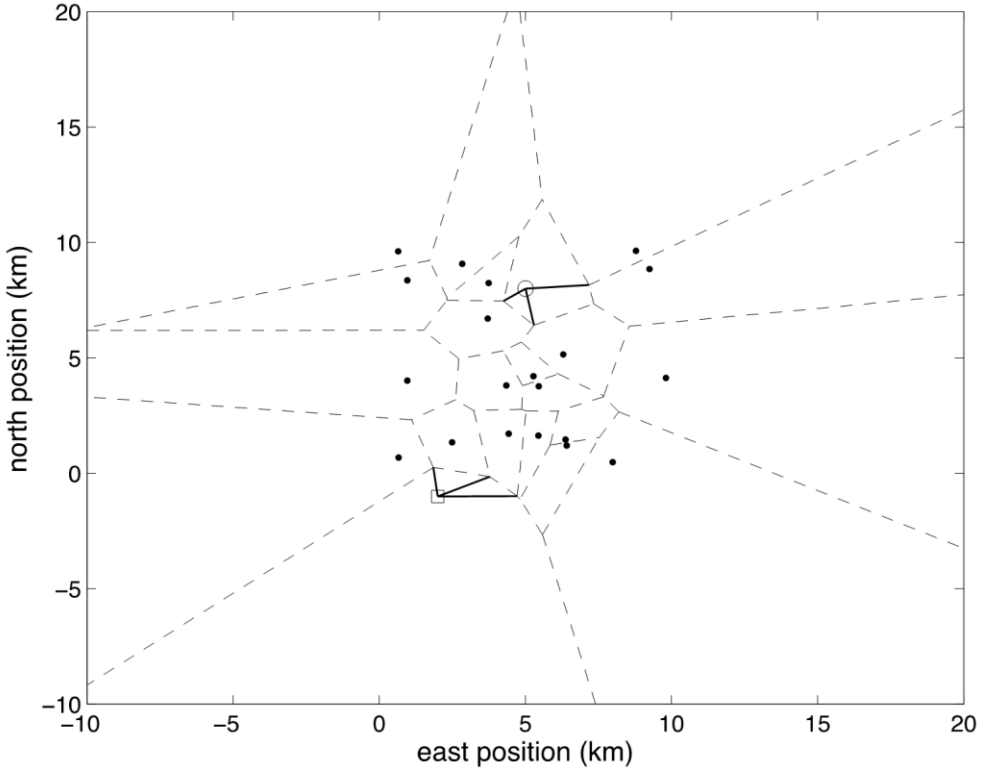
Voronoi Graph Example

- 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
- What is the objective?
  - Shortest path?
  - Furthest from obstacles?

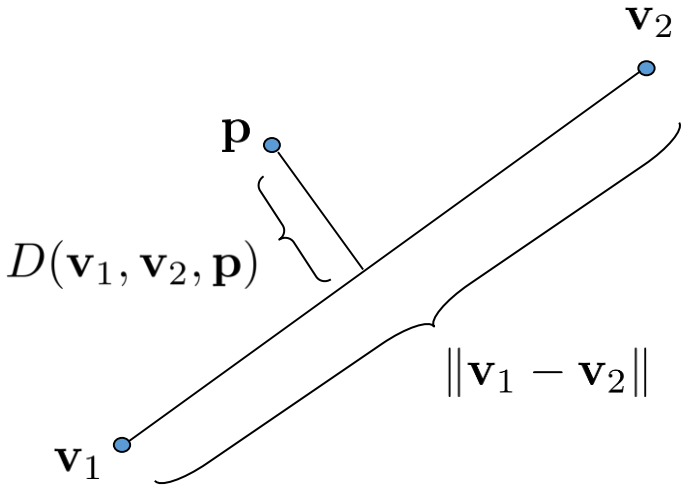


Path Cost Calculation

- 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:

$$\mathbf{w}(\sigma) = (1-\sigma)\mathbf{v}_1 + \sigma\mathbf{v}_2 \quad \sigma \in [0,1]$$

- 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 σ that minimizes D:

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

- Location along edge for which D is minimum:

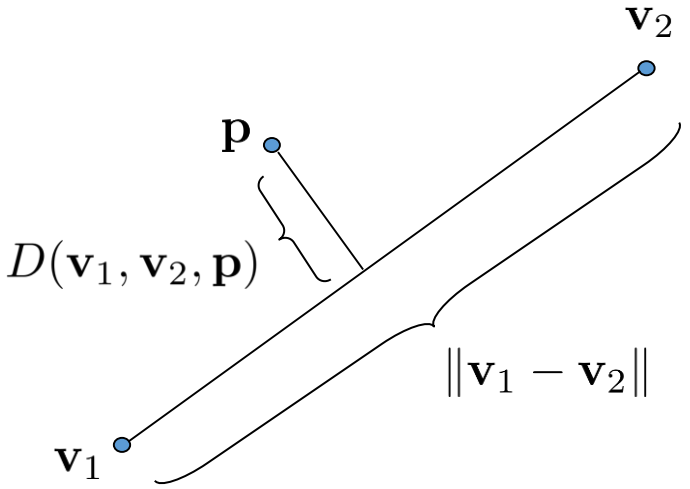
$$\mathbf{w}(\sigma^*) = \sqrt{\|\mathbf{p} - \mathbf{v}_1\|^2 - \frac{((\mathbf{v}_1 - \mathbf{p})^\top (\mathbf{v}_1 - \mathbf{v}_2))^2}{\|\mathbf{v}_1 - \mathbf{v}_2\|^2}}$$

- Define distance to edge for σ* < 0, σ*>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 Q and line segment

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

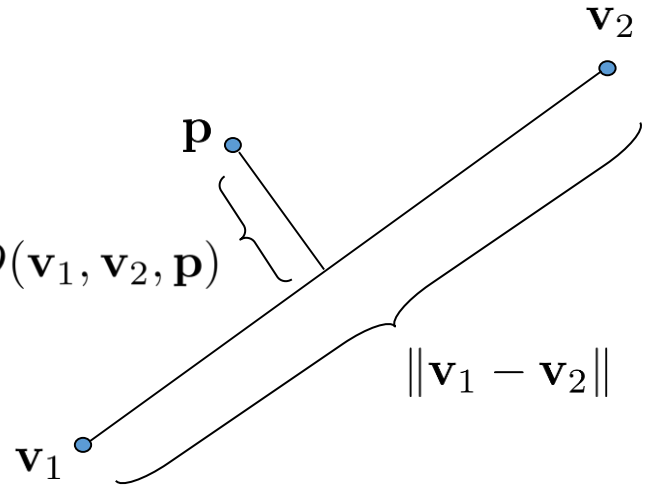


Path Cost Calculation

- Cost for traveling along edge ($\mathbf{v_1}$,$\mathbf{v_2}$) is assigned as:

$$J(\mathbf{v}_1, \mathbf{v}_2) = \underbrace{k_1 \|\mathbf{v}_1 - \mathbf{v}_2\|}_{\text{length of edge}} + \underbrace{\frac{k_2}{D(\mathbf{v}_1, \mathbf{v}_2, \mathcal{Q})}}_{\text{reciprocal of distance to closest point in } \mathcal{Q}}$$

$D(\mathbf{v}_1, \mathbf{v}_2, \mathbf{p})$



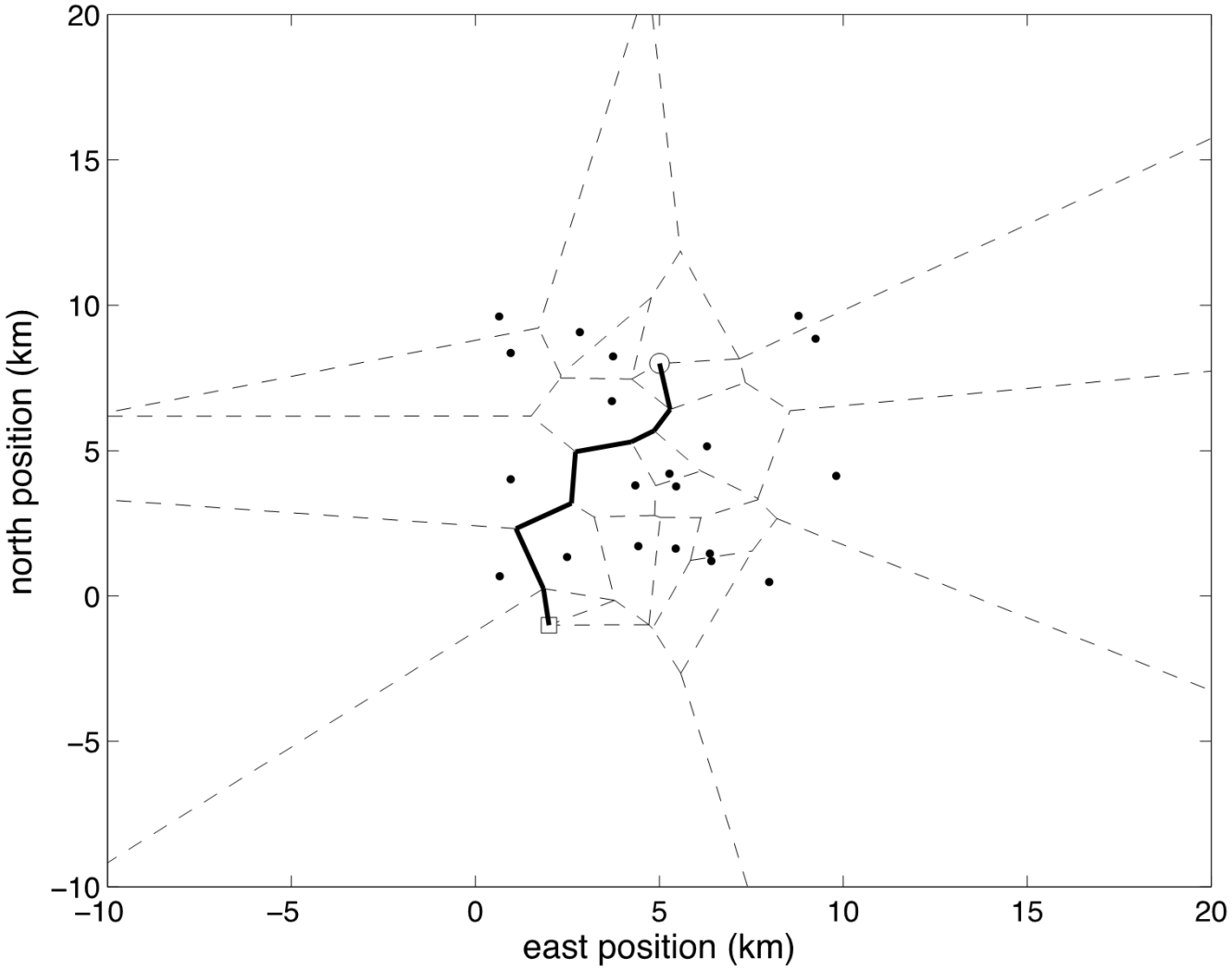
k_1 and k_2 are positive weights

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

Voronoi Path Planning Algorithm

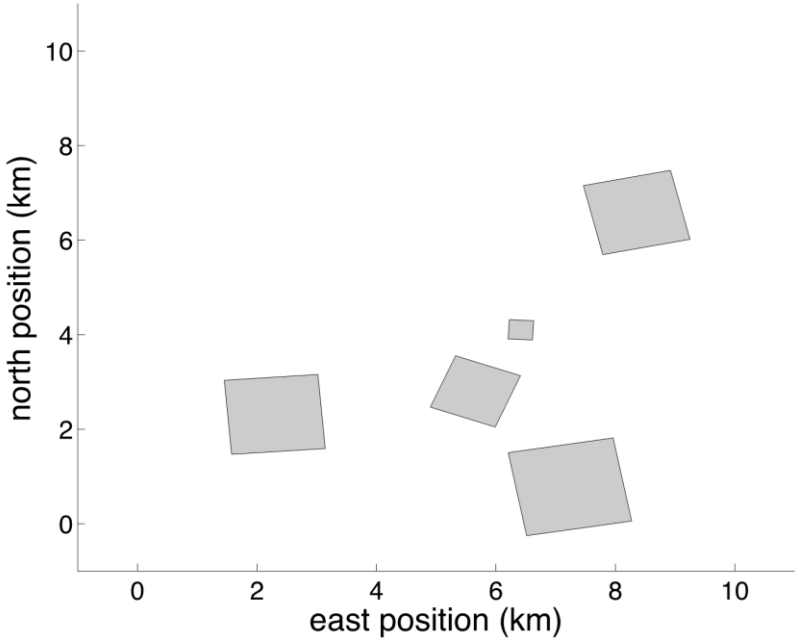
- Plan Voronoi Path: $W = planVoronoi(Q, ps, pe)$
- Input: Obstacle points Q, start position p_s, end position p_e
- Require: $|Q| \geq 10$, randomly add points if necessary
  - $(V, E) = constructVoronoiGraph(Q)$
  - $V^+ = V \cup \{p_s\} \cup \{p_e\}$
  - Find $\{v_{1s}, v_{2s}, v_{3s}\}$, the three closest points in V to p_s, $\{v_{1e}, v_{2e}, v_{3e}\}$, the three closest points in V to p_e
  - $E^+ = E \cup_{i=1,2,3} (v_{is}, ps) \cup_{i=1,2,3} (v_{ie}, pe)$
  - **for each** element $(v_a, v_b) \in E$ do:
    - Assign edge cost $J_{ab} = J(v_a, vb)$
  - **end for**
  - $W = DijkstraSearch(V^+, E^+, J)$
  - return W

Voronoi Path Planning Result



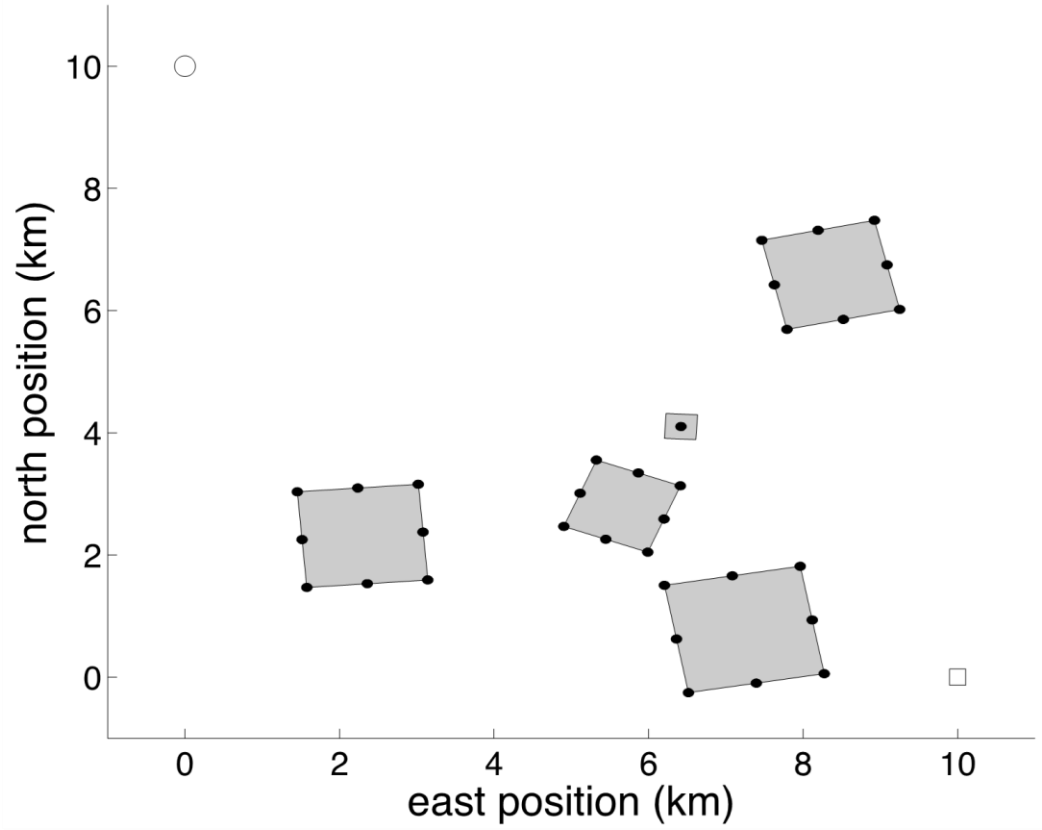
Voronoi Path Planning: Non-point Obstacles

- How can we deal with solid obstacles?
- Point obstacles at center of spatial obstacles won't provide safe path options around obstacles



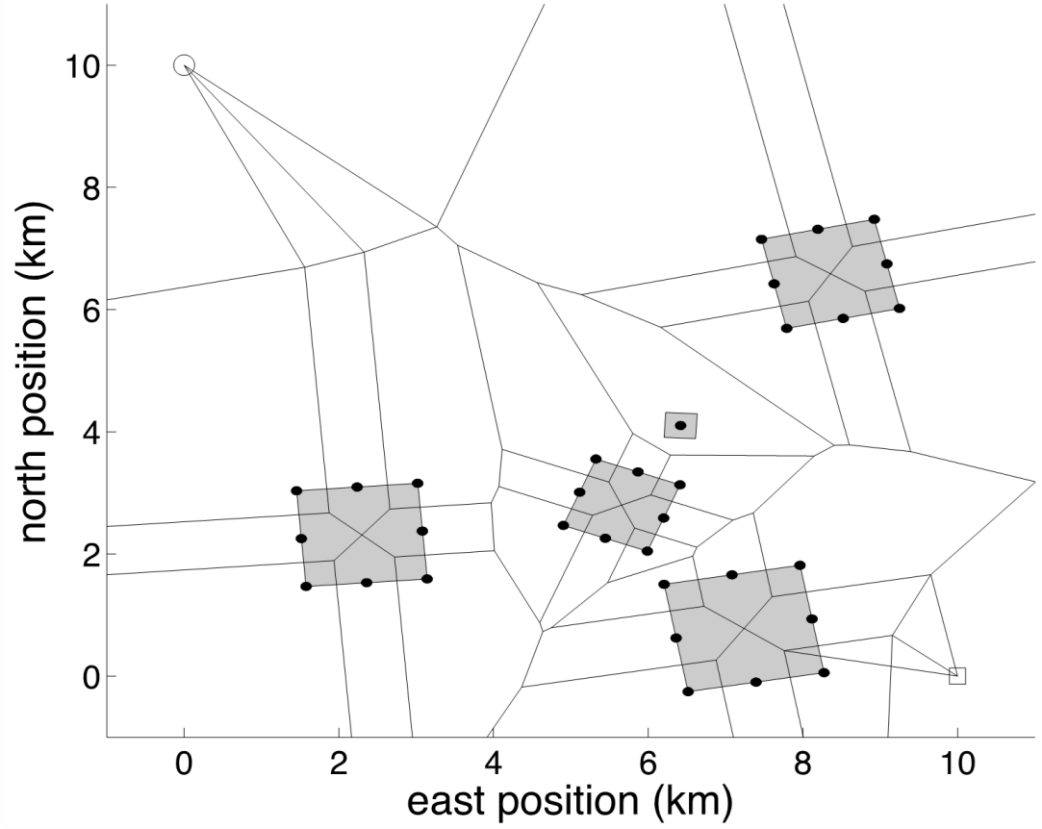
Non-point Obstacles: Step 1

- Insert points around perimeter of obstacles



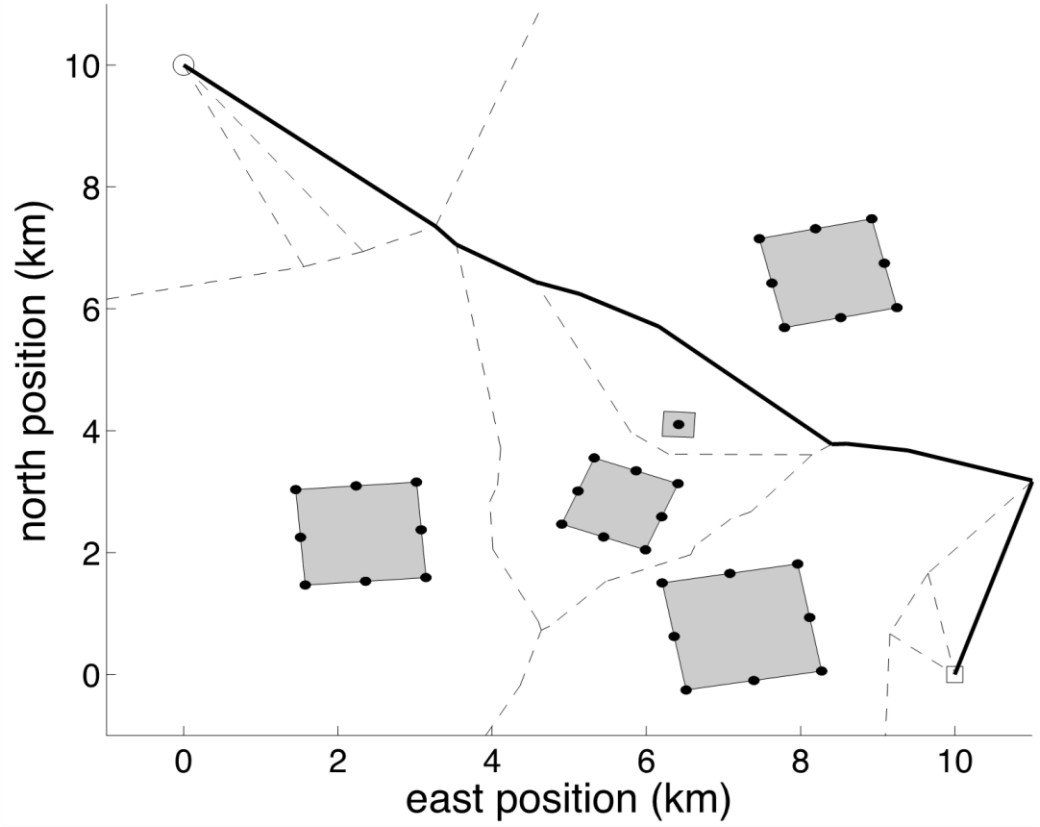
Non-point Obstacles: Step 1

- Construct Voronoi graph
- Non feasible path edges inside obstacles



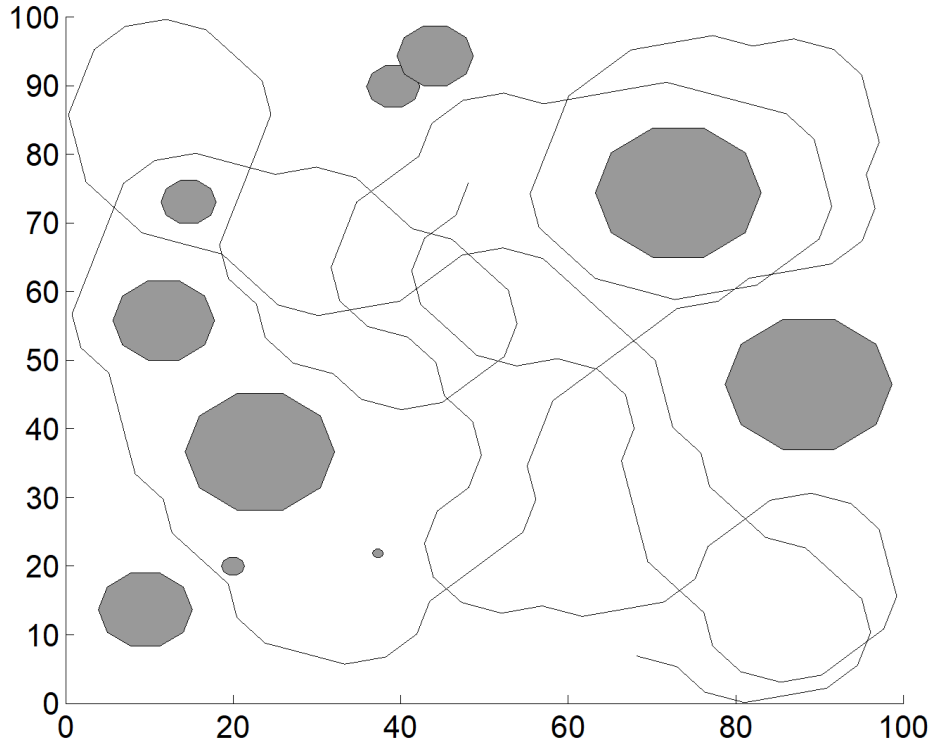
Non-point Obstacles: Step 1

- Remove infeasible path edges from graph
- Search graph for best path



Coverage Planning

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



Coverage Planning

- 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 some 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 Planning Algorithm

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

Coverage Planning Algorithm

- Plan Cover Path: $planCover(T, \boldsymbol{Y}, \boldsymbol{p})$
- Input: Terrain map T, return map $\boldsymbol{Y}$, initial configuration $\boldsymbol{p}_s$
  - Initialize look-ahead tree $G = (V, E)$ as $V = \{\boldsymbol{p}_s\}$, $E = \emptyset$
  - Initialize return map Y = {Yi: i indexes the terrain}
  - $\boldsymbol{p} = \boldsymbol{p}_s$
  - **for each** planning cycle do:
    - $G = generateTree(\boldsymbol{p}, T, \boldsymbol{Y})$
    - $W = highestReturnPath(G)$
    - Update $\boldsymbol{p}$ by moving along the first segment of W
    - Reset $G = (V, E)$ as $V = \{p\}$, $E = \emptyset$
    - $\boldsymbol{Y} = updateReturnMap(\boldsymbol{Y}, \boldsymbol{p})$
  - **end for**

Find out more

- https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
- https://en.wikipedia.org/wiki/A*_search_algorithm


Thank you!
Please ask your question!