Seminar

Path planning using Voronoi diagrams and B-Splines

Stefano Martina
stefano.martina@stud.unifi.it

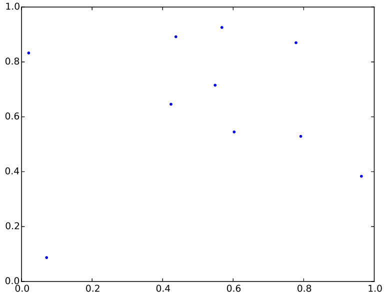
Università degli Studi Firenze

23 may 2016

Voronoi diagrams

Input: A set of points in plane (or space) called sites

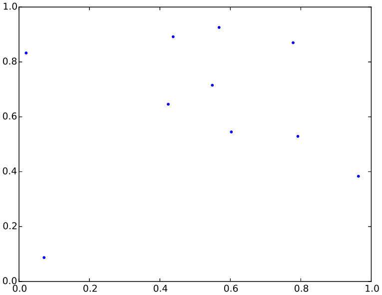
Output: A partition of the plane (or space) such that each point of a region is nearer to a certain site respect to the others

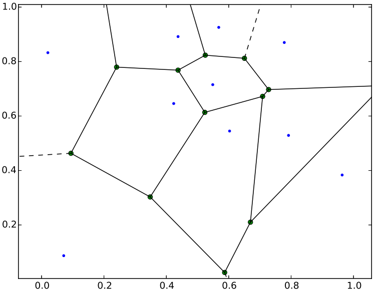


Voronoi diagrams

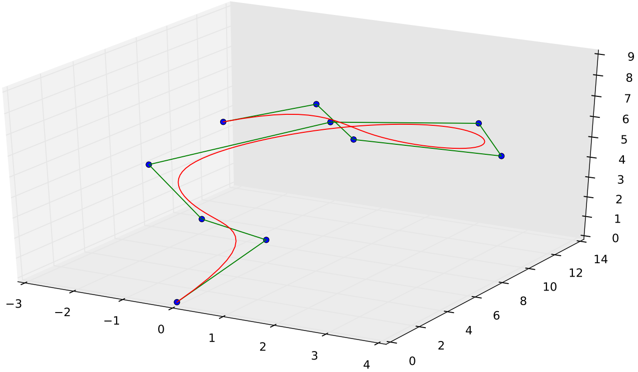
Input: A set of points in plane (or space) called sites

Output: A partition of the plane (or space) such that each point of a region is nearer to a certain site respect to the others



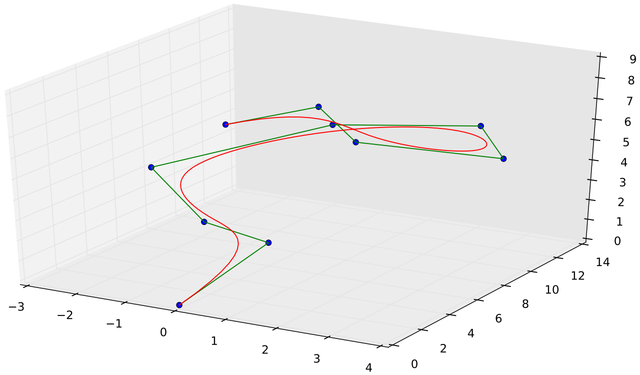


B-spline



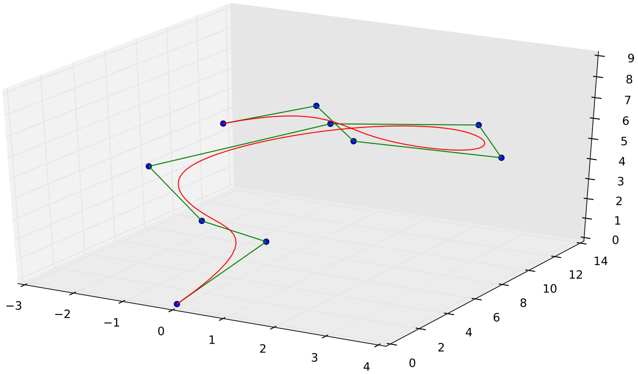
- ✔ parametric curves
- ✔ follow the shape of a control poligon
- ✔ can interpolate the extremes of the control polygon

B-spline



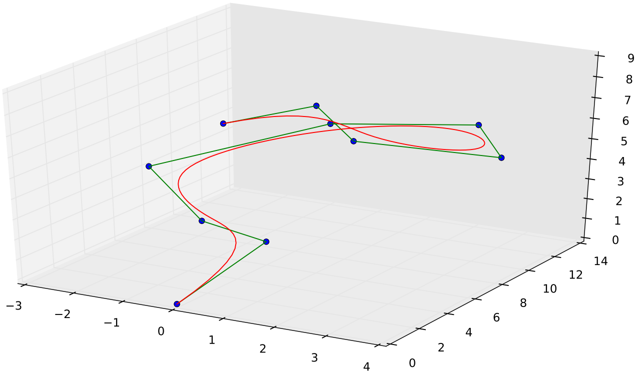
- ✔ parametric curves
- ✔ follow the shape of a control poligon
- ✔ can interpolate the extremes of the control polygon

B-spline



- ✔ parametric curves
- ✔ follow the shape of a control poligon
- ✔ can interpolate the extremes of the control polygon

B-spline



- ✔ parametric curves
- ✔ follow the shape of a control poligon
- ✔ can interpolate the extremes of the control polygon

B-splines details

- ✔ Order k $(= \textit{degree} + 1)$
- ✔ Extended partition (of parametric space $[a, b]$)

$$T = \{t_0, \dots, t_{k-2}, t_{k-1}, \dots, t_{n+1}, t_{n+2}, \dots, t_{n+k}\}$$

$$t_0 \leq \dots \leq t_{k-2} \leq t_{k-1} (\equiv a) < \dots < t_{n+1} (\equiv b) \leq t_{n+2} \leq \dots \leq t_{n+k}$$

- ✔ $n + 1$ basis

$$N_{i,1}(t) = \begin{cases} 1, & \text{if} \quad t_i \leq t < t_{i+1} \\ 0, & \text{otherwise} \end{cases}$$

$$N_{i,k}(t) = \omega_{i,k-1}(t) \cdot N_{i,k-1}(t) \; + \; (1 - \omega_{i+1,k-1}(t)) \cdot N_{i+1,k-1}(t)$$

$$\omega_{i,k}(t) = \frac{t - t_i}{t_{i+k} - t_i}.$$

- ✔ B-spline

$$\mathbf{S}(t) = \sum_{i=0}^{n} \mathbf{v_i} \cdot N_{i,k}(t),$$

B-splines details

- Order k $(= \textit{degree} + 1)$
- Extended partition (of parametric space $[a, b]$)

$$T = \{t_0, \ldots, t_{k-2}, t_{k-1}, \ldots, t_{n+1}, t_{n+2}, \ldots, t_{n+k}\}$$

$$t_0 \leq \cdots \leq t_{k-2} \leq t_{k-1} (\equiv a) < \cdots < t_{n+1} (\equiv b) \leq t_{n+2} \leq \cdots \leq t_{n+k}$$

- $n+1$ basis

$$N_{i,1}(t) = \begin{cases} 1, & \text{if} \quad t_i \leq t < t_{i+1} \\ 0, & \text{otherwise} \end{cases}$$

$$N_{i,k}(t) = \omega_{i,k-1}(t) \cdot N_{i,k-1}(t) \; + \; (1 - \omega_{i+1,k-1}(t)) \cdot N_{i+1,k-1}(t)$$

$$\omega_{i,k}(t) = \frac{t - t_i}{t_{i+k} - t_i}.$$

- B-spline

$$\mathbf{S}(t) = \sum_{i=0}^{n} \mathbf{v_i} \cdot N_{i,k}(t),$$

B-splines details

- Order k $(= \mathit{degree} + 1)$
- Extended partition (of parametric space $[a, b]$)

$$T = \{t_0, \ldots, t_{k-2}, t_{k-1}, \ldots, t_{n+1}, t_{n+2}, \ldots, t_{n+k}\}$$

$$t_0 \leq \cdots \leq t_{k-2} \leq t_{k-1} (\equiv a) < \cdots < t_{n+1} (\equiv b) \leq t_{n+2} \leq \cdots \leq t_{n+k}$$

- $n + 1$ basis

$$N_{i,1}(t) = \begin{cases} 1, & \text{if} \quad t_i \leq t < t_{i+1} \\ 0, & \text{otherwise} \end{cases}$$

$$N_{i,k}(t) = \omega_{i,k-1}(t) \cdot N_{i,k-1}(t) \ + \ (1 - \omega_{i+1,k-1}(t)) \cdot N_{i+1,k-1}(t)$$

$$\omega_{i,k}(t) = \frac{t - t_i}{t_{i+k} - t_i}.$$

- B-spline

$$\mathbf{S}(t) = \sum_{i=0}^{n} \mathbf{v_i} \cdot N_{i,k}(t),$$

B-splines details

- ✔ Order k ($= degree + 1$)
- ✔ Extended partition (of parametric space $[a, b]$)

$$T = \{t_0, \dots, t_{k-2}, t_{k-1}, \dots, t_{n+1}, t_{n+2}, \dots, t_{n+k}\}$$

$$t_0 \leq \dots \leq t_{k-2} \leq t_{k-1} (\equiv a) < \dots < t_{n+1} (\equiv b) \leq t_{n+2} \leq \dots \leq t_{n+k}$$

- ✔ $n+1$ basis

$$N_{i,1}(t) = \begin{cases} 1, & \text{if} \quad t_i \leq t < t_{i+1} \\ 0, & \text{otherwise} \end{cases}$$

$$N_{i,k}(t) = \omega_{i,k-1}(t) \cdot N_{i,k-1}(t) \; + \; (1 - \omega_{i+1,k-1}(t)) \cdot N_{i+1,k-1}(t)$$

$$\omega_{i,k}(t) = \frac{t - t_i}{t_{i+k} - t_i}.$$

- ✔ B-spline

$$\mathbf{S}(t) = \sum_{i=0}^{n} \mathbf{v_i} \cdot N_{i,k}(t),$$

Useful properties of B-spline curves

- ✔ Clamped if $t_0 = \cdots = t_{k-1}$ and $t_{n+1} = \cdots = t_{n+k}$
- ✔ Continuity C^{k-2} between polynomials (or C^{m-1})
- ✔ Contained inside the union of convex hulls composed of consecutive k vertexes of control polygon

- ✔ Touch to the segment between $k-1$ aligned control vertexes
- ✔ Lay down in the segment between k aligned control vertexes

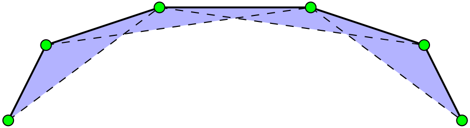
Useful properties of B-spline curves

- Clamped if $t_0 = \cdots = t_{k-1}$ and $t_{n+1} = \cdots = t_{n+k}$
- Continuity C^{k-2} between polynomials (or C^{m-1})
- Contained inside the union of convex hulls composed of consecutive k vertexes of control polygon

- Touch to the segment between $k-1$ aligned control vertexes
- Lay down in the segment between k aligned control vertexes

Useful properties of B-spline curves

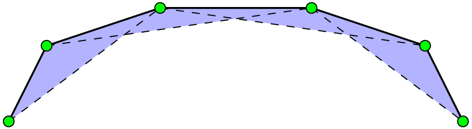
- ✔ Clamped if $t_0 = \cdots = t_{k-1}$ and $t_{n+1} = \cdots = t_{n+k}$
- ✔ Continuity C^{k-2} between polynomials (or C^{m-1})
- ✔ Contained inside the union of convex hulls composed of consecutive k vertexes of control polygon



- ✔ Touch to the segment between $k-1$ aligned control vertexes
- ✔ Lay down in the segment between k aligned control vertexes

Useful properties of B-spline curves

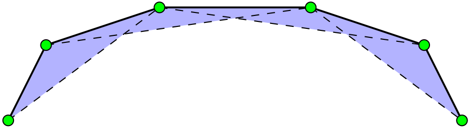
- ✔ Clamped if $t_0 = \cdots = t_{k-1}$ and $t_{n+1} = \cdots = t_{n+k}$
- ✔ Continuity C^{k-2} between polynomials (or C^{m-1})
- ✔ Contained inside the union of convex hulls composed of consecutive k vertexes of control polygon



- ✔ Touch to the segment between $k-1$ aligned control vertexes
- ✔ Lay down in the segment between k aligned control vertexes

Useful properties of B-spline curves

- Clamped if $t_0 = \cdots = t_{k-1}$ and $t_{n+1} = \cdots = t_{n+k}$
- Continuity C^{k-2} between polynomials (or C^{m-1})
- Contained inside the union of convex hulls composed of consecutive k vertexes of control polygon



- Touch to the segment between $k-1$ aligned control vertexes
- Lay down in the segment between k aligned control vertexes

Dijkstra algorithm

```
def dijkstra(graph, start, end):
  path = []
  Q = priorityQueue.PQueue()
  dist = {}
  prev = {}
  for node in graph.nodes(): #populate the queue
    if node != start:
      dist[node] = inf
      Q.add(node, inf)
    else:
      dist[node] = 0
      Q.add(node, 0)
  while True:  #main loop
    u = Q.pop() #take nearest node and remove from queue
    if u == end or dist[u] == inf: #finished (good or bad)
      break
    #all neighbors still in queue
    for v in Q.filterGet(lambda node: node in graph.neighbors(u)):
      tmpDist = dist[u] + graph[u][v]['weight']
      if tmpDist < dist[v]: #if distance shorter update values
        dist[v] = tmpDist
        prev[v] = u
        Q.add(v, tmpDist) #update distance also in queue
  u = end
  while u in prev:  #backward recreation of path
      u = prev[u]
      path[:0] = [u]
  if path:
      path[len(path):] = [end]
      path[:0] = [start]
  return path
```

Background

Main problem

Path planning from a start point to an end point in 3D space with obstacles using Voronoi diagrams.

1. Distribute points in the surfaces of obstacles
   - and optionally in the surface of bounding box
2. Build Voronoi diagram using those points as source
3. Transform the Voronoi diagram in a graph
   - cells vertexes as nodes
   - cells edges as arcs (infinite edges ignored)
4. Prune the arcs that crosses an obstacle's surface
5. Attach the start and end points to the graph as nodes
6. Calculate the shortest path from start node to end node using Dijkstra's algorithm.

Background

Main problem

Path planning from a start point to an end point in 3D space with obstacles using Voronoi diagrams.

1. Distribute points in the surfaces of obstacles
   - and optionally in the surface of bounding box
2. Build Voronoi diagram using those points as source
3. Transform the Voronoi diagram in a graph
   - cells vertexes as nodes
   - cells edges as arcs (infinite edges ignored)
4. Prune the arcs that crosses an obstacle's surface
5. Attach the start and end points to the graph as nodes
6. Calculate the shortest path from start node to end node using Dijkstra's algorithm.

Background

Main problem

Path planning from a start point to an end point in 3D space with obstacles using Voronoi diagrams.

1. Distribute points in the surfaces of obstacles
   - and optionally in the surface of bounding box
2. Build Voronoi diagram using those points as source
3. Transform the Voronoi diagram in a graph
   - cells vertexes as nodes
   - cells edges as arcs (infinite edges ignored)
4. Prune the arcs that crosses an obstacle's surface
5. Attach the start and end points to the graph as nodes
6. Calculate the shortest path from start node to end node using Dijkstra's algorithm.

Background

Main problem

Path planning from a start point to an end point in 3D space with obstacles using Voronoi diagrams.

1. Distribute points in the surfaces of obstacles
   - and optionally in the surface of bounding box
2. Build Voronoi diagram using those points as source
3. Transform the Voronoi diagram in a graph
   - cells vertexes as nodes
   - cells edges as arcs (infinite edges ignored)
4. Prune the arcs that crosses an obstacle's surface
5. Attach the start and end points to the graph as nodes
6. Calculate the shortest path from start node to end node using Dijkstra's algorithm.

Background

Main problem

Path planning from a start point to an end point in 3D space with obstacles using Voronoi diagrams.

1. Distribute points in the surfaces of obstacles
   - and optionally in the surface of bounding box
2. Build Voronoi diagram using those points as source
3. Transform the Voronoi diagram in a graph
   - cells vertexes as nodes
   - cells edges as arcs (infinite edges ignored)
4. Prune the arcs that crosses an obstacle's surface
5. Attach the start and end points to the graph as nodes
6. Calculate the shortest path from start node to end node using Dijkstra's algorithm.

Background

Main problem

Path planning from a start point to an end point in 3D space with obstacles using Voronoi diagrams.

1. Distribute points in the surfaces of obstacles
   - and optionally in the surface of bounding box
2. Build Voronoi diagram using those points as source
3. Transform the Voronoi diagram in a graph
   - cells vertexes as nodes
   - cells edges as arcs (infinite edges ignored)
4. Prune the arcs that crosses an obstacle's surface
5. Attach the start and end points to the graph as nodes
6. Calculate the shortest path from start node to end node using Dijkstra's algorithm.

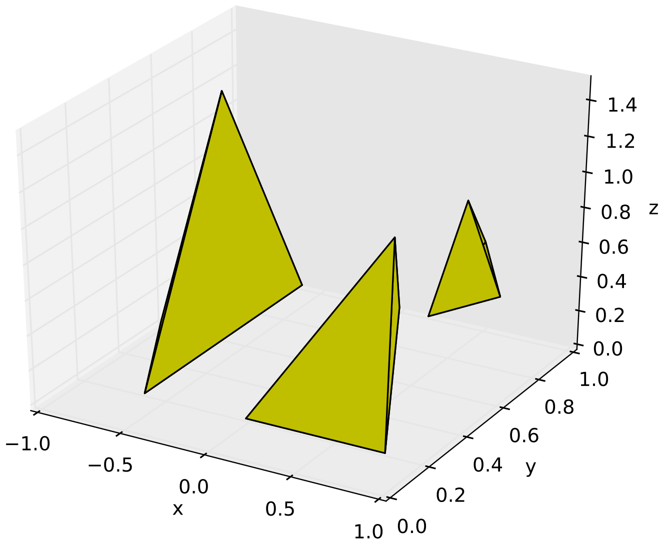
Background

Main problem

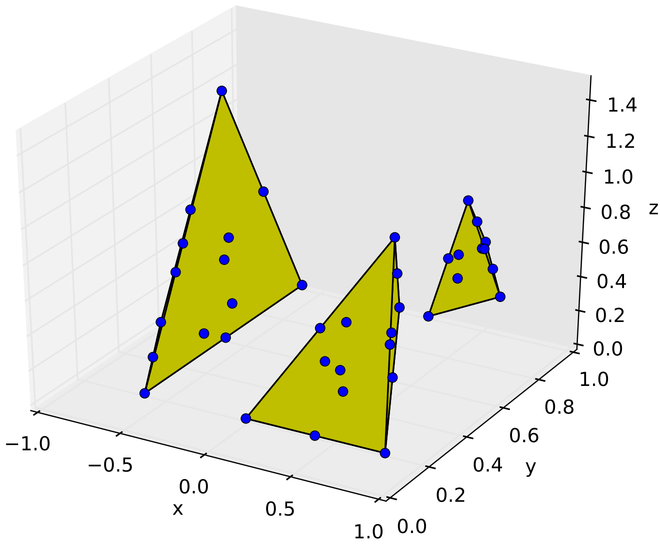
Path planning from a start point to an end point in 3D space with obstacles using Voronoi diagrams.

1. Distribute points in the surfaces of obstacles
   - and optionally in the surface of bounding box
2. Build Voronoi diagram using those points as source
3. Transform the Voronoi diagram in a graph
   - cells vertexes as nodes
   - cells edges as arcs (infinite edges ignored)
4. Prune the arcs that crosses an obstacle's surface
5. Attach the start and end points to the graph as nodes
6. Calculate the shortest path from start node to end node using Dijkstra's algorithm.

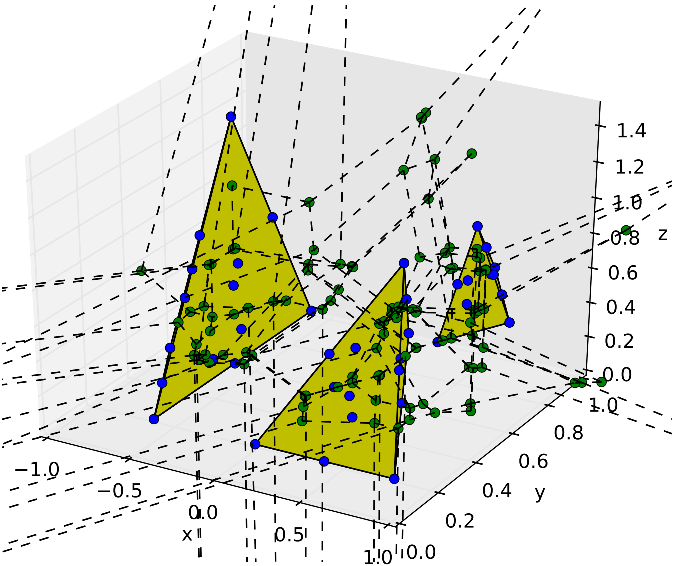
Example



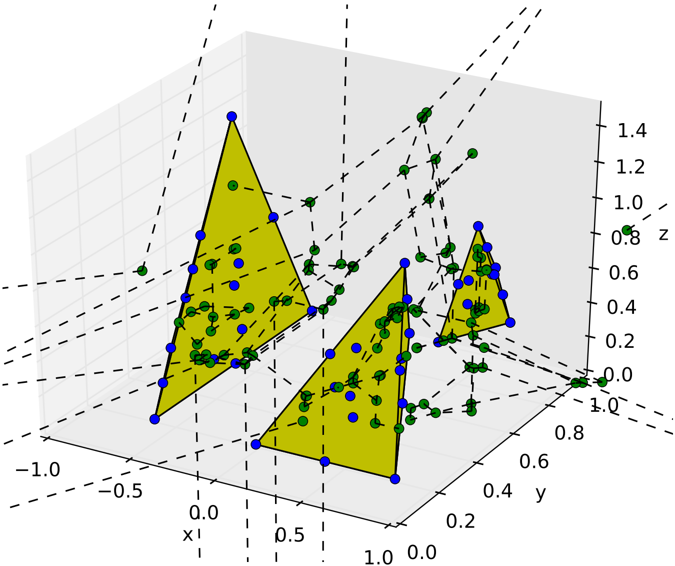
Example



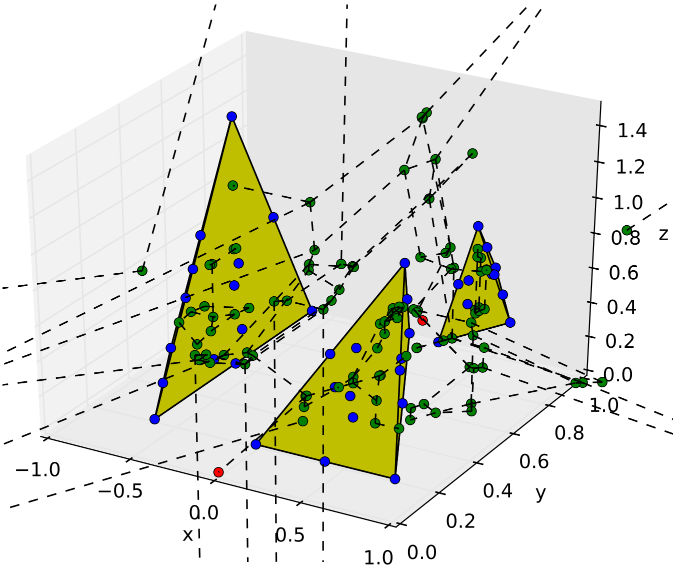
Example



Example



Example



Improvement

Idea

Make a smoother curve instead of finding the polygonal chain of the shortest path in the structure

- ✔ we can use a B-Spline that
  - interpolate the start and end vertexes
  - use the shortest path found with Dijkstra as control polygon

Improvement

Idea

Make a smoother curve instead of finding the polygonal chain of the shortest path in the structure

✔ we can use a B-Spline that

- interpolate the start and end vertexes
- use the shortest path found with Dijkstra as control polygon

Improvement

Idea

Make a smoother curve instead of finding the polygonal chain of the shortest path in the structure

- ✔ we can use a B-Spline that
  - interpolate the start and end vertexes
  - use the shortest path found with Dijkstra as control polygon

Improvement

Idea

Make a smoother curve instead of finding the polygonal chain of the shortest path in the structure

- ✔ we can use a B-Spline that
  - interpolate the start and end vertexes
  - use the shortest path found with Dijkstra as control polygon

Problem

✔ The control polygon is free from obstacles by construction

- (the graph is pruned of the arcs that cross an obstacle's surface)

✔ But the curve is not guaranteed to be free from obstacles

Problem

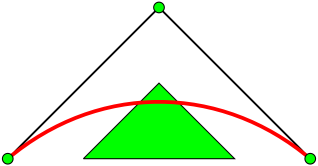
- ✔ The control polygon is free from obstacles by construction
  - (the graph is pruned of the arcs that cross an obstacle's surface)
- ✔ But the curve is not guaranteed to be free from obstacles

Problem

- ✔ The control polygon is free from obstacles by construction
  - (the graph is pruned of the arcs that cross an obstacle's surface)
- ✔ But the curve is not guaranteed to be free from obstacles

Problem

- ✔ The control polygon is free from obstacles by construction
  - (the graph is pruned of the arcs that cross an obstacle's surface)
- ✔ But the curve is not guaranteed to be free from obstacles



Solution

✔ A B-Spline of order k is contained inside the union of convex hulls composed of consecutive k vertexes of control polygon

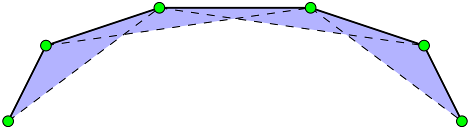
Idea

✔ we can use a quadratic B-Spline (grade 2, order 3) to smooth the path

✔ and keep triangles formed by three consecutive points free from obstacles

Solution

- ✔ A B-Spline of order k is contained inside the union of convex hulls composed of consecutive k vertexes of control polygon

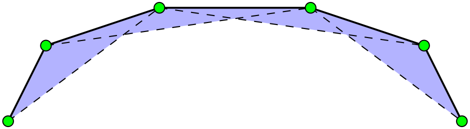


Idea

- ✔ we can use a quadratic B-Spline (grade 2, order 3) to smooth the path
- ✔ and keep triangles formed by three consecutive points free from obstacles

Solution

✔ A B-Spline of order k is contained inside the union of convex hulls composed of consecutive k vertexes of control polygon



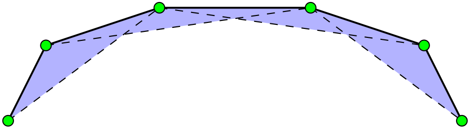
Idea

✔ we can use a quadratic B-Spline (grade 2, order 3) to smooth the path

✔ and keep triangles formed by three consecutive points free from obstacles

Solution

- ✔ A B-Spline of order k is contained inside the union of convex hulls composed of consecutive k vertexes of control polygon



Idea

- ✔ we can use a quadratic B-Spline (grade 2, order 3) to smooth the path
- ✔ and keep triangles formed by three consecutive points free from obstacles

First implementation

A variation of Dijkstra algorithm is developed where:

1. create an ordered triple for each three consecutive nodes in the graph
2. check if the triangle corresponding to each triple intersect an obstacle
3. populate the priority queue with obstacle free triples
   - the initial weight is 0 for triples where the first node is the start node
   - is ∞ otherwise
4. pop the triple with lowest weight from the priority queue
5. update the weight and pointer to previous of all neighbouring triples
   - a triple B is subsequent to a triple A if $(A[2] = B[1]) \wedge (A[3] = B[2])$
   - the weight of a neighbour is $W(B) = W(A) + dist(A[1], A[2])$
6. repeat from point 4 until popped a special ending triple or a triple with weight ∞
7. the shortest path (with free triangular convex hull) can be obtained following the previous pointer from the ending triple, and deconstructing the triples

First implementation

A variation of Dijkstra algorithm is developed where:

1. create an ordered triple for each three consecutive nodes in the graph
2. check if the triangle corresponding to each triple intersect an obstacle
3. populate the priority queue with obstacle free triples
   - the initial weight is 0 for triples where the first node is the start node
   - is ∞ otherwise
4. pop the triple with lowest weight from the priority queue
5. update the weight and pointer to previous of all neighbouring triples
   - a triple B is subsequent to a triple A if $(A[2] = B[1]) \wedge (A[3] = B[2])$
   - the weight of a neighbour is $W(B) = W(A) + dist(A[1], A[2])$
6. repeat from point 4 until popped a special ending triple or a triple with weight ∞
7. the shortest path (with free triangular convex hull) can be obtained following the previous pointer from the ending triple, and deconstructing the triples

First implementation

A variation of Dijkstra algorithm is developed where:

1. create an ordered triple for each three consecutive nodes in the graph
2. check if the triangle corresponding to each triple intersect an obstacle
3. populate the priority queue with obstacle free triples
   - the initial weight is 0 for triples where the first node is the start node
   - is ∞ otherwise
4. pop the triple with lowest weight from the priority queue
5. update the weight and pointer to previous of all neighbouring triples
   - a triple B is subsequent to a triple A if $(A[2] = B[1]) \wedge (A[3] = B[2])$
   - the weight of a neighbour is $W(B) = W(A) + dist(A[1], A[2])$
6. repeat from point 4 until popped a special ending triple or a triple with weight ∞
7. the shortest path (with free triangular convex hull) can be obtained following the previous pointer from the ending triple, and deconstructing the triples

First implementation

A variation of Dijkstra algorithm is developed where:

1. create an ordered triple for each three consecutive nodes in the graph
2. check if the triangle corresponding to each triple intersect an obstacle
3. populate the priority queue with obstacle free triples
    - the initial weight is 0 for triples where the first node is the start node
    - is ∞ otherwise
4. pop the triple with lowest weight from the priority queue
5. update the weight and pointer to previous of all neighbouring triples
    - a triple B is subsequent to a triple A if $(A[2] = B[1]) \wedge (A[3] = B[2])$
    - the weight of a neighbour is $W(B) = W(A) + dist(A[1], A[2])$
6. repeat from point 4 until popped a special ending triple or a triple with weight ∞
7. the shortest path (with free triangular convex hull) can be obtained following the previous pointer from the ending triple, and deconstructing the triples

First implementation

A variation of Dijkstra algorithm is developed where:

1. create an ordered triple for each three consecutive nodes in the graph
2. check if the triangle corresponding to each triple intersect an obstacle
3. populate the priority queue with obstacle free triples
   - the initial weight is 0 for triples where the first node is the start node
   - is ∞ otherwise
4. pop the triple with lowest weight from the priority queue
5. update the weight and pointer to previous of all neighbouring triples
   - a triple B is subsequent to a triple A if $(A[2] = B[1]) \wedge (A[3] = B[2])$
   - the weight of a neighbour is $W(B) = W(A) + dist(A[1], A[2])$
6. repeat from point 4 until popped a special ending triple or a triple with weight ∞
7. the shortest path (with free triangular convex hull) can be obtained following the previous pointer from the ending triple, and deconstructing the triples

First implementation

A variation of Dijkstra algorithm is developed where:

1. create an ordered triple for each three consecutive nodes in the graph
2. check if the triangle corresponding to each triple intersect an obstacle
3. populate the priority queue with obstacle free triples
    - the initial weight is 0 for triples where the first node is the start node
    - is ∞ otherwise
4. pop the triple with lowest weight from the priority queue
5. update the weight and pointer to previous of all neighbouring triples
    - a triple B is subsequent to a triple A if $(A[2] = B[1]) \wedge (A[3] = B[2])$
    - the weight of a neighbour is $W(B) = W(A) + dist(A[1], A[2])$
6. repeat from point 4 until popped a special ending triple or a triple with weight ∞
7. the shortest path (with free triangular convex hull) can be obtained following the previous pointer from the ending triple, and deconstructing the triples

First implementation

A variation of Dijkstra algorithm is developed where:

1. create an ordered triple for each three consecutive nodes in the graph
2. check if the triangle corresponding to each triple intersect an obstacle
3. populate the priority queue with obstacle free triples
   - the initial weight is 0 for triples where the first node is the start node
   - is ∞ otherwise
4. pop the triple with lowest weight from the priority queue
5. update the weight and pointer to previous of all neighbouring triples
   - a triple B is subsequent to a triple A if $(A[2] = B[1]) \wedge (A[3] = B[2])$
   - the weight of a neighbour is $W(B) = W(A) + dist(A[1], A[2])$
6. repeat from point 4 until popped a special ending triple or a triple with weight ∞
7. the shortest path (with free triangular convex hull) can be obtained following the previous pointer from the ending triple, and deconstructing the triples

First implementation

A variation of Dijkstra algorithm is developed where:

1. create an ordered triple for each three consecutive nodes in the graph
2. check if the triangle corresponding to each triple intersect an obstacle
3. populate the priority queue with obstacle free triples
    - the initial weight is 0 for triples where the first node is the start node
    - is ∞ otherwise
4. pop the triple with lowest weight from the priority queue
5. update the weight and pointer to previous of all neighbouring triples
    - a triple B is subsequent to a triple A if $(A[2] = B[1]) \wedge (A[3] = B[2])$
    - the weight of a neighbour is $W(B) = W(A) + dist(A[1], A[2])$
6. repeat from point 4 until popped a special ending triple or a triple with weight ∞
7. the shortest path (with free triangular convex hull) can be obtained following the previous pointer from the ending triple, and deconstructing the triples

First implementation

A variation of Dijkstra algorithm is developed where:

1. create an ordered triple for each three consecutive nodes in the graph
2. check if the triangle corresponding to each triple intersect an obstacle
3. populate the priority queue with obstacle free triples
   - the initial weight is 0 for triples where the first node is the start node
   - is ∞ otherwise
4. pop the triple with lowest weight from the priority queue
5. update the weight and pointer to previous of all neighbouring triples
   - a triple B is subsequent to a triple A if $(A[2] = B[1]) \wedge (A[3] = B[2])$
   - the weight of a neighbour is $W(B) = W(A) + dist(A[1], A[2])$
6. repeat from point 4 until popped a special ending triple or a triple with weight ∞
7. the shortest path (with free triangular convex hull) can be obtained following the previous pointer from the ending triple, and deconstructing the triples

First implementation

A variation of Dijkstra algorithm is developed where:

1. create an ordered triple for each three consecutive nodes in the graph
2. check if the triangle corresponding to each triple intersect an obstacle
3. populate the priority queue with obstacle free triples
   - the initial weight is 0 for triples where the first node is the start node
   - is ∞ otherwise
4. pop the triple with lowest weight from the priority queue
5. update the weight and pointer to previous of all neighbouring triples
   - a triple B is subsequent to a triple A if $(A[2] = B[1]) \wedge (A[3] = B[2])$
   - the weight of a neighbour is $W(B) = W(A) + dist(A[1], A[2])$
6. repeat from point 4 until popped a special ending triple or a triple with weight ∞
7. the shortest path (with free triangular convex hull) can be obtained following the previous pointer from the ending triple, and deconstructing the triples

First implementation

A variation of Dijkstra algorithm is developed where:

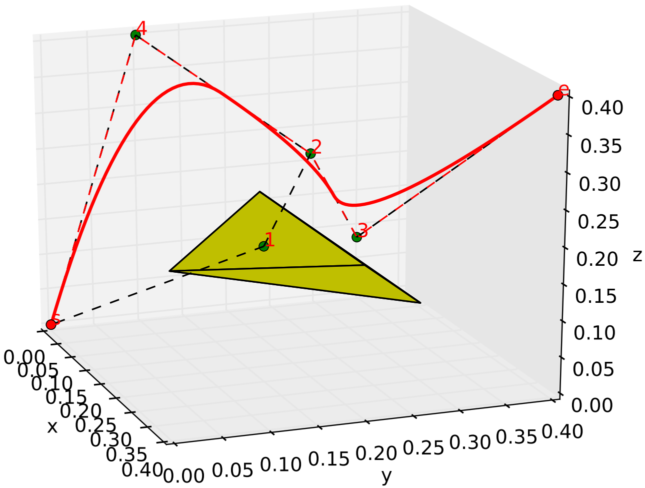
1. create an ordered triple for each three consecutive nodes in the graph
2. check if the triangle corresponding to each triple intersect an obstacle
3. populate the priority queue with obstacle free triples
   - the initial weight is 0 for triples where the first node is the start node
   - is ∞ otherwise
4. pop the triple with lowest weight from the priority queue
5. update the weight and pointer to previous of all neighbouring triples
   - a triple B is subsequent to a triple A if $(A[2] = B[1]) \wedge (A[3] = B[2])$
   - the weight of a neighbour is $W(B) = W(A) + dist(A[1], A[2])$
6. repeat from point 4 until popped a special ending triple or a triple with weight ∞
7. the shortest path (with free triangular convex hull) can be obtained following the previous pointer from the ending triple, and deconstructing the triples

First implementation

A variation of Dijkstra algorithm is developed where:

1. create an ordered triple for each three consecutive nodes in the graph
2. check if the triangle corresponding to each triple intersect an obstacle
3. populate the priority queue with obstacle free triples
   - the initial weight is 0 for triples where the first node is the start node
   - is ∞ otherwise
4. pop the triple with lowest weight from the priority queue
5. update the weight and pointer to previous of all neighbouring triples
   - a triple B is subsequent to a triple A if $(A[2] = B[1]) \wedge (A[3] = B[2])$
   - the weight of a neighbour is $W(B) = W(A) + dist(A[1], A[2])$
6. repeat from point 4 until popped a special ending triple or a triple with weight ∞
7. the shortest path (with free triangular convex hull) can be obtained following the previous pointer from the ending triple, and deconstructing the triples

Example


4
2
1
3
e
s
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
z
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
x
0.00 0.05 0.10 0.15 0.20 0.25 0.30 0.35 0.40
y

Complexity

Total (if obstacle area is not a function of the number of obstacles)

$\mathcal{O}(d^2|Ob|^2 + d^3|Ob|)$

- ✔ The predominant factor is for checking the triples collisions with obstacles
  - if d is constant $\mathcal{O}(|Ob|^2)$
- ✔ But if we focus only on routing (i.e. we construct the graph only once)
  - if d is constant $\mathcal{O}(|Ob| \log |Ob|)$
  - (same of Dijkstra with fixed degree)

Complexity

Total (if obstacle area is not a function of the number of obstacles)

$\mathcal{O}(d^2|Ob|^2 + d^3|Ob|)$

- ✔ The predominant factor is for checking the triples collisions with obstacles
  - if d is constant $\mathcal{O}(|Ob|^2)$
- ✔ But if we focus only on routing (i.e. we construct the graph only once)
  - if d is constant $\mathcal{O}(|Ob| \log |Ob|)$
  - (same of Dijkstra with fixed degree)

Complexity

Total (if obstacle area is not a function of the number of obstacles)

$\mathcal{O}(d^2|Ob|^2 + d^3|Ob|)$

- ✔ The predominant factor is for checking the triples collisions with obstacles
  - if d is constant $\mathcal{O}(|Ob|^2)$
- ✔ But if we focus only on routing (i.e. we construct the graph only once)
  - if d is constant $\mathcal{O}(|Ob| \log |Ob|)$
  - (same of Dijkstra with fixed degree)

Complexity

Total (if obstacle area is not a function of the number of obstacles)

$\mathcal{O}(d^2|Ob|^2 + d^3|Ob|)$

- ✔ The predominant factor is for checking the triples collisions with obstacles
  - if d is constant $\mathcal{O}(|Ob|^2)$
- ✔ But if we focus only on routing (i.e. we construct the graph only once)
  - if d is constant $\mathcal{O}(|Ob| \log |Ob|)$
  - (same of Dijkstra with fixed degree)

Complexity

Total (if obstacle area is not a function of the number of obstacles)

$\mathcal{O}(d^2|Ob|^2 + d^3|Ob|)$

- ✔ The predominant factor is for checking the triples collisions with obstacles
  - if d is constant $\mathcal{O}(|Ob|^2)$
- ✔ But if we focus only on routing (i.e. we construct the graph only once)
  - if d is constant $\mathcal{O}(|Ob| \log |Ob|)$
  - (same of Dijkstra with fixed degree)

Complexity

Total (if obstacle area is not a function of the number of obstacles)

$\mathcal{O}(d^2|Ob|^2 + d^3|Ob|)$

- ✔ The predominant factor is for checking the triples collisions with obstacles
  - if d is constant $\mathcal{O}(|Ob|^2)$
- ✔ But if we focus only on routing (i.e. we construct the graph only once)
  - if d is constant $\mathcal{O}(|Ob| \log |Ob|)$
  - (same of Dijkstra with fixed degree)

Second implementation

Reason

- ✔ First implementation interesting for complexity analysis
- ✘ but rejects many paths

- ✔ Add aligned control vertexes when an obstacle intersect a triple

- ✔ new vertexes calculated as linear combination of other vertexes using coefficients from obstacle detection

Second implementation

Reason

- ✔ First implementation interesting for complexity analysis
- ✘ but rejects many paths

- ✔ Add aligned control vertexes when an obstacle intersect a triple

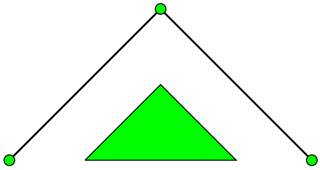
- ✔ new vertexes calculated as linear combination of other vertexes using coefficients from obstacle detection

Second implementation

Reason

- ✔ First implementation interesting for complexity analysis
- ✘ but rejects many paths

✔ Add aligned control vertexes when an obstacle intersect a triple



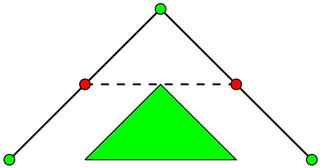
✔ new vertexes calculated as linear combination of other vertexes using coefficients from obstacle detection

Second implementation

Reason

- ✔ First implementation interesting for complexity analysis
- ✘ but rejects many paths

- ✔ Add aligned control vertexes when an obstacle intersect a triple



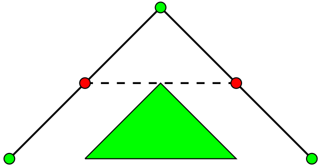
- ✔ new vertexes calculated as linear combination of other vertexes using coefficients from obstacle detection

Second implementation

Reason

- ✔ First implementation interesting for complexity analysis
- ✘ but rejects many paths

- ✔ Add aligned control vertexes when an obstacle intersect a triple



- ✔ new vertexes calculated as linear combination of other vertexes using coefficients from obstacle detection

Increase degree

Continuity

- ✔ Using quadratic B-Splines means C^1 continuity
- ✘ Not nice

- ✘ If we simply increase the B-Spline degree $\rightarrow$ convex hull is not planar anymore
  - convex hull is formed of union of tetrahedra

Solution

- ✔ Add aligned vertexes in control polygon
  - and then increase the degree

Increase degree

Continuity

✔ Using quadratic B-Splines means C^1 continuity

✘ Not nice

✘ If we simply increase the B-Spline degree $\rightarrow$ convex hull is not planar anymore

- convex hull is formed of union of tetrahedra

Solution

✔ Add aligned vertexes in control polygon

- and then increase the degree

Increase degree

Continuity

- ✔ Using quadratic B-Splines means C^1 continuity
- ✘ Not nice

- ✘ If we simply increase the B-Spline degree $\rightarrow$ convex hull is not planar anymore
  - convex hull is formed of union of tetrahedra

Solution

- ✔ Add aligned vertexes in control polygon
  - and then increase the degree

Increase degree

Continuity

- ✔ Using quadratic B-Splines means C^1 continuity
- ✘ Not nice

- ✘ If we simply increase the B-Spline degree $\rightarrow$ convex hull is not planar anymore
  - convex hull is formed of union of tetrahedra

Solution

- ✔ Add aligned vertexes in control polygon
  - and then increase the degree

Increase degree

Continuity

- ✔ Using quadratic B-Splines means C^1 continuity
- ✘ Not nice

- ✘ If we simply increase the B-Spline degree $\rightarrow$ convex hull is not planar anymore
  - convex hull is formed of union of tetrahedra

Solution

- ✔ Add aligned vertexes in control polygon
  - and then increase the degree

Increase degree

Continuity

- ✔ Using quadratic B-Splines means C^1 continuity
- ✘ Not nice

✘ If we simply increase the B-Spline degree → convex hull is not planar anymore

- convex hull is formed of union of tetrahedra

Solution

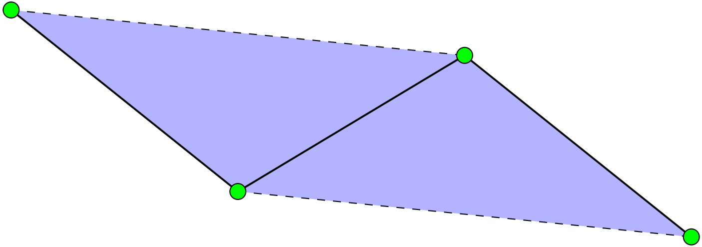
- ✔ Add aligned vertexes in control polygon
  - and then increase the degree

Example: quadratic to quartic (k=3 → k=5)

✔ Add 2 vertexes per edge

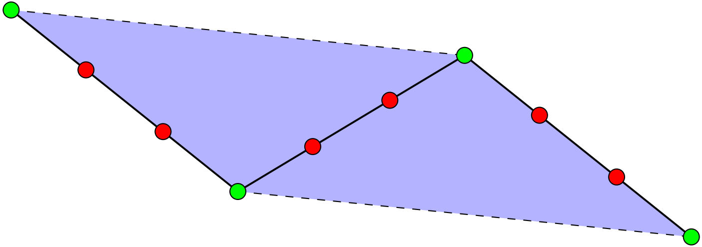
Example: quadratic to quartic (k=3 → k=5)

- ✔ Add 2 vertexes per edge



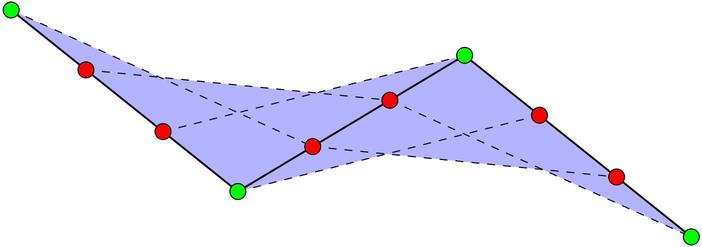
Example: quadratic to quartic (k=3 → k=5)

✔ Add 2 vertexes per edge



Example: quadratic to quartic (k=3 → k=5)

✔ Add 2 vertexes per edge



Postprocessing

Purpose

- ✔ Simplify the control polygon
- ✔ Remove useless turns

- ✔ After Dijkstra
- ✔ For each triple (a, b, c) of consecutive points in path
- ✔ If no obstacles intersect the triangle $\rightarrow$ the triple is simplified to a single edge (a, c)

☛ After simplification, new neighbouring triples need to be obstacle-free

Postprocessing

Purpose

- ✔ Simplify the control polygon
- ✔ Remove useless turns

- ✔ After Dijkstra
- ✔ For each triple (a, b, c) of consecutive points in path
- ✔ If no obstacles intersect the triangle $\rightarrow$ the triple is simplified to a single edge (a, c)

☛ After simplification, new neighbouring triples need to be obstacle-free

Postprocessing

Purpose

- ✔ Simplify the control polygon
- ✔ Remove useless turns

- ✔ After Dijkstra
- ✔ For each triple (a, b, c) of consecutive points in path
- ✔ If no obstacles intersect the triangle → the triple is simplified to a single edge (a, c)

☛ After simplification, new neighbouring triples need to be obstacle-free

Postprocessing

Purpose

- ✔ Simplify the control polygon
- ✔ Remove useless turns

- ✔ After Dijkstra
- ✔ For each triple (a, b, c) of consecutive points in path
- ✔ If no obstacles intersect the triangle $\rightarrow$ the triple is simplified to a single edge (a, c)

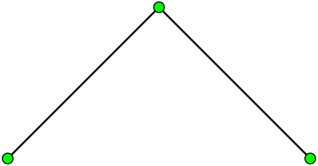
☛ After simplification, new neighbouring triples need to be obstacle-free

Postprocessing

Purpose

- ✔ Simplify the control polygon
- ✔ Remove useless turns

- ✔ After Dijkstra
- ✔ For each triple (a, b, c) of consecutive points in path
- ✔ If no obstacles intersect the triangle $\rightarrow$ the triple is simplified to a single edge (a, c)
- ☛ After simplification, new neighbouring triples need to be obstacle-free

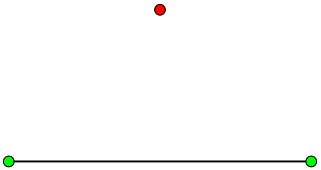


Postprocessing

Purpose

- ✔ Simplify the control polygon
- ✔ Remove useless turns

- ✔ After Dijkstra
- ✔ For each triple (a, b, c) of consecutive points in path
- ✔ If no obstacles intersect the triangle $\rightarrow$ the triple is simplified to a single edge (a, c)
- ☛ After simplification, new neighbouring triples need to be obstacle-free

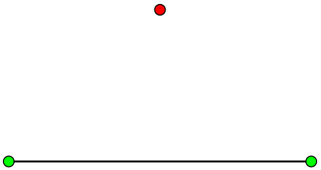


Postprocessing

Purpose

- ✔ Simplify the control polygon
- ✔ Remove useless turns

- ✔ After Dijkstra
- ✔ For each triple (a, b, c) of consecutive points in path
- ✔ If no obstacles intersect the triangle $\rightarrow$ the triple is simplified to a single edge (a, c)
- ☛ After simplification, new neighbouring triples need to be obstacle-free



Used technologies

Used technologies

python 3

SciPy

Used technologies

python 3

SciPy

Visualization Toolkit

Future improvements

- Change underlying structure
  - visibility graph
  - rapidly exploring random tree (RRT)
  - other ...
- Improve postprocessing
  - make a symmetric algorithm (path from a to b = path from b to a)
- Make optimization process
  - try to find the best path that satisfy some constraints
    - max curvature
    - max torsion
    - others ...

Future improvements

- ✔ Change underlying structure
  - visibility graph
  - rapidly exploring random tree (RRT)
  - other ...
- ✔ Improve postprocessing
  - make a symmetric algorithm (path from a to b = path from b to a)
- ✔ Make optimization process
  - try to find the best path that satisfy some constraints
    - ⋆ max curvature
    - ⋆ max torsion
    - ⋆ others ...

Future improvements

- ✔ Change underlying structure
  - visibility graph
  - rapidly exploring random tree (RRT)
  - other . . .
- ✔ Improve postprocessing
  - make a symmetric algorithm (path from a to b = path from b to a)
- ✔ Make optimization process
  - try to find the best path that satisfy some constraints
    - ⋆ max curvature
    - ⋆ max torsion
    - ⋆ others . . .

Future improvements

- ✔ Change underlying structure
  - visibility graph
  - rapidly exploring random tree (RRT)
  - other ...
- ✔ Improve postprocessing
  - make a symmetric algorithm (path from a to b = path from b to a)
- ✔ Make optimization process
  - try to find the best path that satisfy some constraints
    - ★ max curvature
    - ★ max torsion
    - ★ others ...

Future improvements

- ✔ Change underlying structure
  - visibility graph
  - rapidly exploring random tree (RRT)
  - other ...
- ✔ Improve postprocessing
  - make a symmetric algorithm (path from a to b = path from b to a)
- ✔ Make optimization process
  - try to find the best path that satisfy some constraints
    - ⋆ max curvature
    - ⋆ max torsion
    - ⋆ others ...

Future improvements

- ✔ Change underlying structure
  - visibility graph
  - rapidly exploring random tree (RRT)
  - other ...
- ✔ Improve postprocessing
  - make a symmetric algorithm (path from a to b = path from b to a)
- ✔ Make optimization process
  - try to find the best path that satisfy some constraints
    - ⋆ max curvature
    - ⋆ max torsion
    - ⋆ others ...

Future improvements

- ✔ Change underlying structure
  - visibility graph
  - rapidly exploring random tree (RRT)
  - other ...
- ✔ Improve postprocessing
  - make a symmetric algorithm (path from a to b = path from b to a)
- ✔ Make optimization process
  - try to find the best path that satisfy some constraints
    - max curvature
    - max torsion
    - others ...

Future improvements

- ✔ Change underlying structure
  - visibility graph
  - rapidly exploring random tree (RRT)
  - other ...
- ✔ Improve postprocessing
  - make a symmetric algorithm (path from a to b = path from b to a)
- ✔ Make optimization process
  - try to find the best path that satisfy some constraints
    - ★ max curvature
    - ★ max torsion
    - ★ others ...

Future improvements

- ✔ Change underlying structure
  - visibility graph
  - rapidly exploring random tree (RRT)
  - other ...
- ✔ Improve postprocessing
  - make a symmetric algorithm (path from a to b = path from b to a)
- ✔ Make optimization process
  - try to find the best path that satisfy some constraints
    - ⋆ max curvature
    - ⋆ max torsion
    - ⋆ others ...

Future improvements

- ✔ Change underlying structure
  - visibility graph
  - rapidly exploring random tree (RRT)
  - other ...
- ✔ Improve postprocessing
  - make a symmetric algorithm (path from a to b = path from b to a)
- ✔ Make optimization process
  - try to find the best path that satisfy some constraints
    - ★ max curvature
    - ★ max torsion
    - ★ others ...

Future improvements

- ✔ Change underlying structure
  - visibility graph
  - rapidly exploring random tree (RRT)
  - other . . .
- ✔ Improve postprocessing
  - make a symmetric algorithm (path from a to b = path from b to a)
- ✔ Make optimization process
  - try to find the best path that satisfy some constraints
    - ⋆ max curvature
    - ⋆ max torsion
    - ⋆ others . . .

The End.

Questions? Thank you!

The End.

Questions? *Thank you!*

The End.

Questions? Thank you!