

Trijkstra

A Dijkstra algorithm application to path planning

Stefano MARTINA

stefano.martina@stud.unifi.it



UNIVERSITÀ
DEGLI STUDI
FIRENZE

4 December 2015

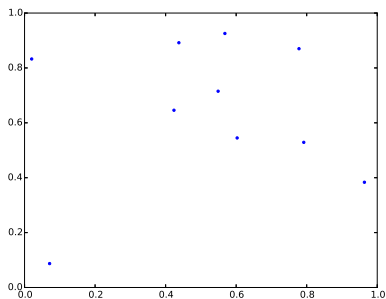


This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

Voronoi diagrams

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

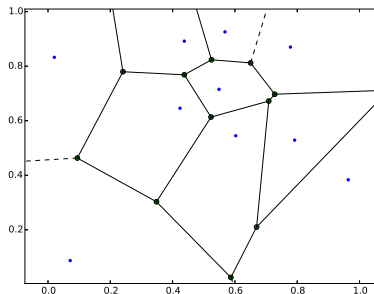
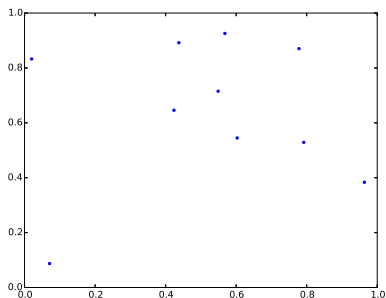
Output: A partition of the plane (or space) such that each point of a **region** is nearer to a certain site respect to the others



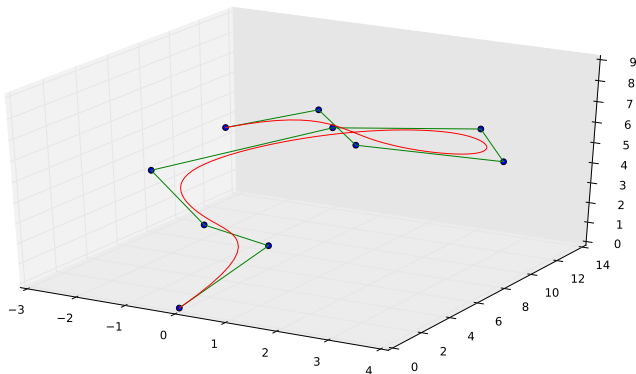
Voronoi diagrams

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

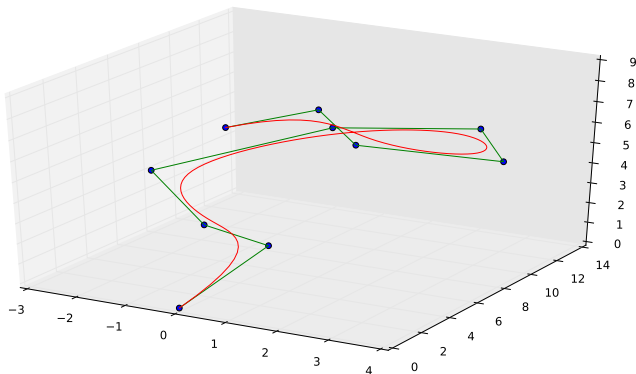


B-spline



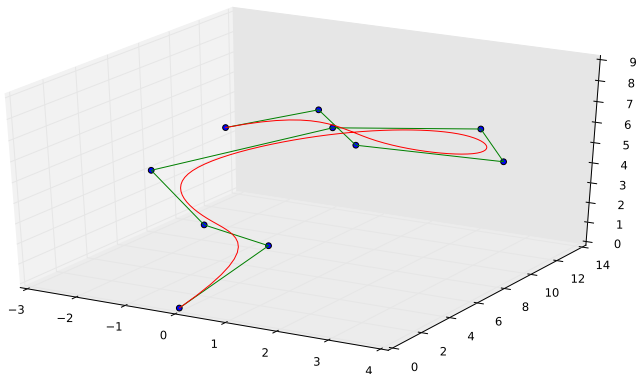
- ✓ parametric curves
- ✓ follow the shape of a control polygon
- ✓ can interpolate the extremes of the control polygon

B-spline



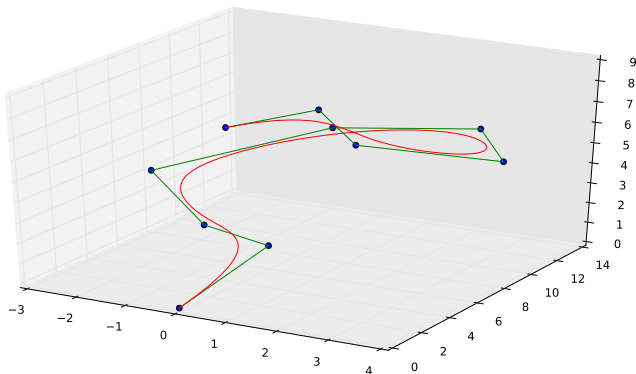
- ✓ **parametric** curves
- ✓ follow the shape of a **control polygon**
- ✓ can interpolate the **extremes** of the control polygon

B-spline



- ✓ **parametric** curves
- ✓ follow the shape of a **control polygon**
- ✓ can interpolate the **extremes** of the control polygon

B-spline



- ✓ **parametric** curves
- ✓ follow the shape of a **control polygon**
- ✓ can interpolate the **extremes** of the control polygon

Dijkstra algorithm

```
1 def dijkstra(graph, start, end):
2     path = []
3     Q = priorityQueue.PQueue()
4     dist = {}
5     prev = {}
6     for node in graph.nodes(): #populate the queue
7         if node != start:
8             dist[node] = inf
9             Q.add(node, inf)
10        else:
11            dist[node] = 0
12            Q.add(node, 0)
13    while True: #main loop
14        u = Q.pop() #take nearest node and remove from queue
15        if u == end or dist[u] == inf: #finished (good or bad)
16            break
17        #all neighbors still in queue
18        for v in Q.filterGet(lambda node: node in graph.neighbors(u)):
19            tmpDist = dist[u] + graph[u][v]['weight']
20            if tmpDist < dist[v]: #if distance shorter update values
21                dist[v] = tmpDist
22                prev[v] = u
23                Q.add(v, tmpDist) #update distance also in queue
24    u = end
25    while u in prev: #backward recreation of path
26        u = prev[u]
27        path[:0] = [u]
28    if path:
29        path[len(path):] = [end]
30        path[:0] = [start]
31    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.

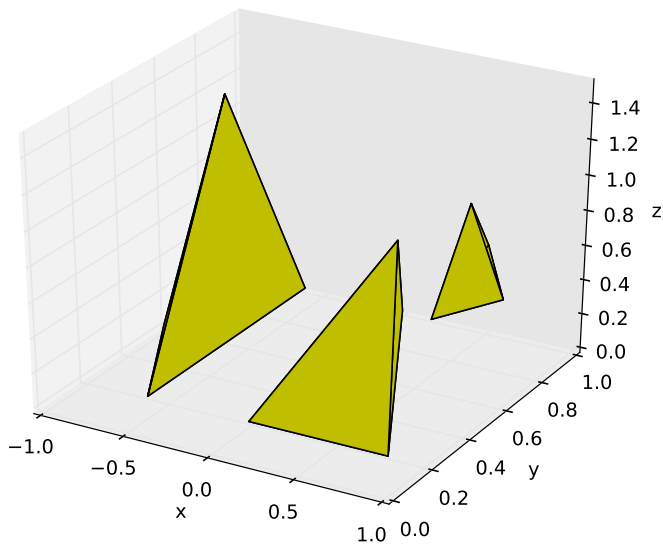
Background

Main problem

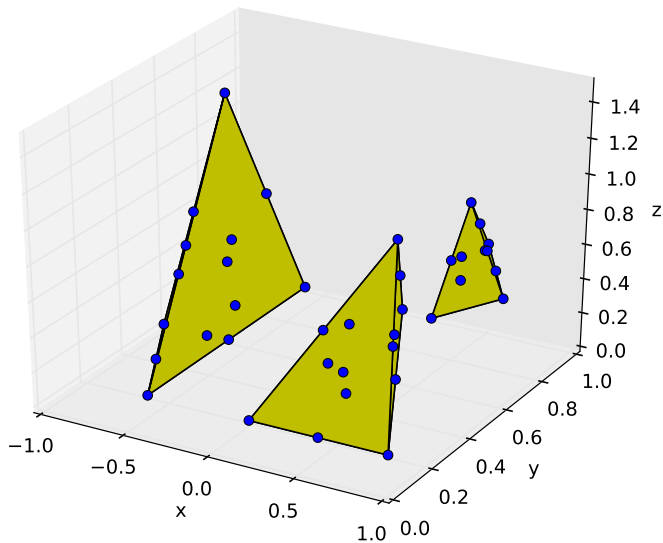
Path planning from a **start** point to an **end** point in 3D space with obstacles using **Voronoi** diagrams.

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.

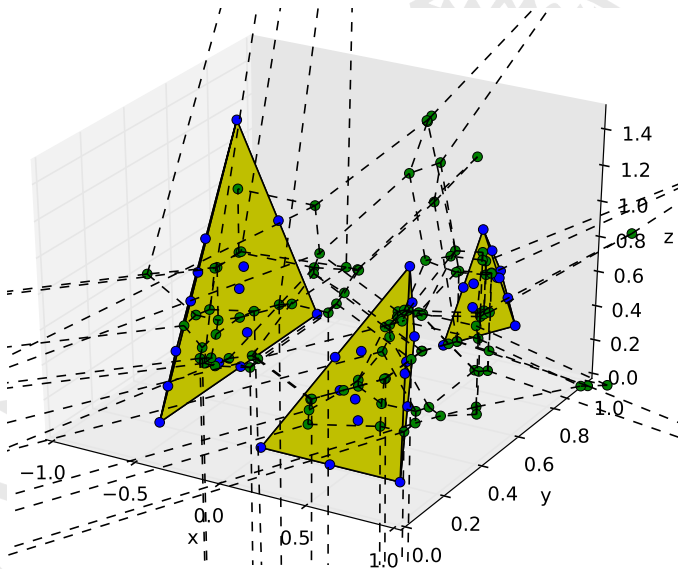
Example



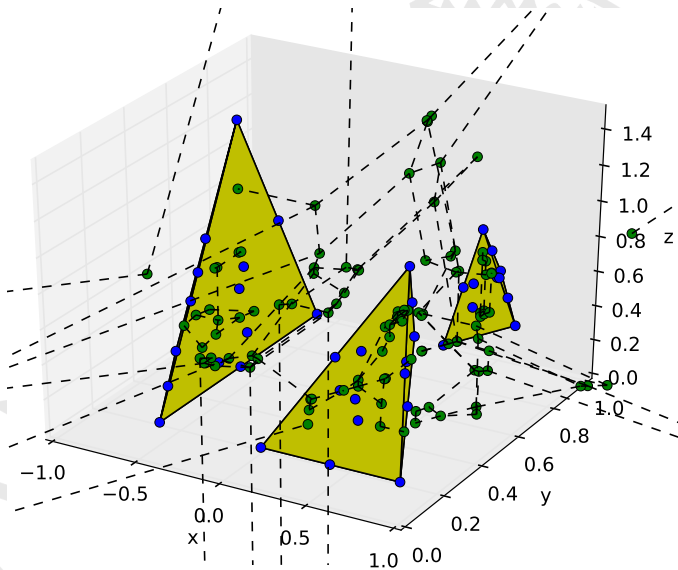
Example



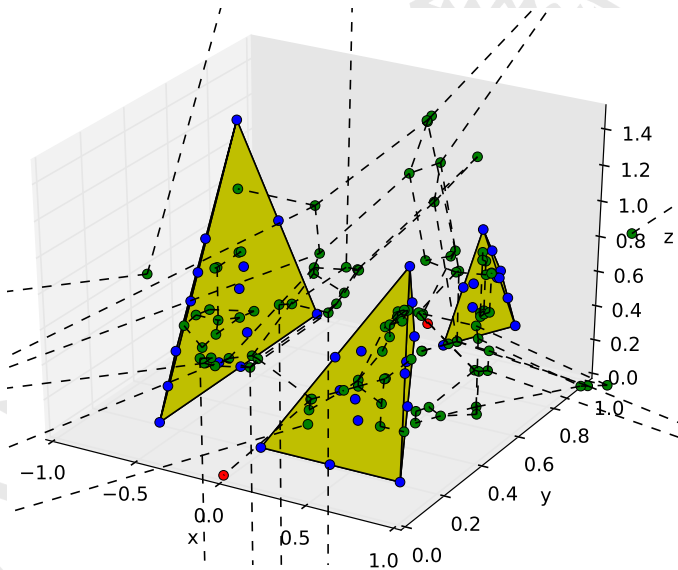
Example



Example



Example



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



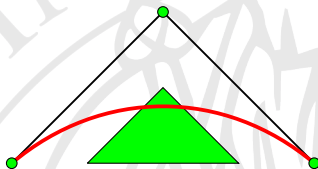
Problem

- ✓ 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 n is contained inside the union of **convex hulls** composed of consecutive n vertexes of control polygon

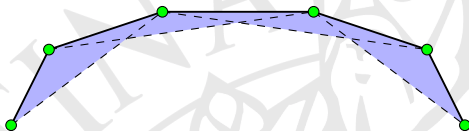


Idea

- ✓ 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 n is contained inside the union of **convex hulls** composed of consecutive n vertexes of control polygon

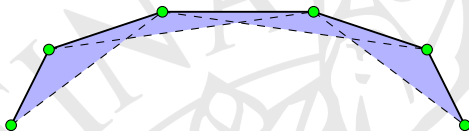


Idea

- ✓ 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 n is contained inside the union of **convex hulls** composed of consecutive n vertexes of control polygon

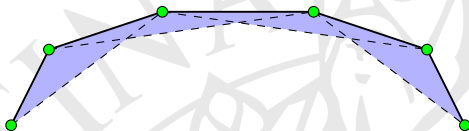


Idea

- ✓ 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 n is contained inside the union of **convex hulls** composed of consecutive n 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

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 $W(B) = W(A) + \text{dist}(A[1], A[2])$
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

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 $W(B) = W(A) + \text{dist}(A[1], A[2])$
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

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 $W(B) = W(A) + \text{dist}(A[1], A[2])$
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

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 $W(B) = W(A) + \text{dist}(A[1], A[2])$
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

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 $W(B) = W(A) + \text{dist}(A[1], A[2])$
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

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 $W(B) = W(A) + \text{dist}(A[1], A[2])$
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

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 $W(B) = W(A) + \text{dist}(A[1], A[2])$
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

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 $W(B) = W(A) + \text{dist}(A[1], A[2])$
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

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 $W(B) = W(A) + \text{dist}(A[1], A[2])$
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

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 $W(B) = W(A) + \text{dist}(A[1], A[2])$
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

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 $W(B) = W(A) + \text{dist}(A[1], A[2])$
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

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 $W(B) = W(A) + \text{dist}(A[1], A[2])$
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

Declarations & Triples creation

```
1 def _trijkstra(self, startA, endA):
2     start = tuple(startA)
3     end = tuple(endA)
4     endTriplet = (end, end, end) #special triplet for termination
5     inf = float("inf")
6     path = []
7     Q = priorityQueue.PQueue()
8     dist = {}
9     prev = {}
10    hits = []

1 for node0 in self._graph.nodes():
2     for node1 in self._graph.neighbors(node0):
3         for node2 in filter(lambda node: node!=node0, self._graph.neighbors(node1)):
4             triplet = (node0, node1, node2)
5             if not triplet[:-1] in hits:
6                 if not self._triangleIntersectPolyhedrons(np.array(node0), np.array(node1),
7                     ↪ np.array(node2)):
8                     if node0 != start:
9                         dist[triplet] = inf
10                        Q.add(triplet, inf)
11                    else:
12                        dist[triplet] = 0
13                        Q.add(triplet, 0)
14                else:
15                    hits[:0] = [triplet]
16
17    dist[endTriplet] = inf
18    Q.add(endTriplet, inf)
```

Main loop

```
1 while True:
2     u = Q.pop()
3
4     if u == endTriplet or dist[u] == inf:
5         break
6
7     for v in Q.filterGet(lambda tri: u[1] == tri[0] and u[2] == tri[1]):
8         tmpDist = dist[u] + self._graph[u[0]][u[1]]['weight']
9         if tmpDist < dist[v]:
10             dist[v] = tmpDist
11             prev[v] = u
12             Q.add(v, tmpDist)
13
14     if u[2] == end:
15         tmpDist = dist[u] + self._graph[u[0]][u[1]]['weight'] +
16             ↪ self._graph[u[1]][u[2]]['weight']
17         if tmpDist < dist[endTriplet]:
18             dist[endTriplet] = tmpDist
19             prev[endTriplet] = u
20             Q.add(v, tmpDist)
```

Path creation

```
1  u = endTriplet
2  while u in prev:
3      u = prev[u]
4      path[:0] = [u[1]]
5
6  if path:
7      path[len(path):] = [end]
8      path[:0] = [start]
9
10 return np.array(path)
```

After

we can use the returned `path` as a `control polygon` for a quadratic B-Spline without problems, and construct a smoother path.

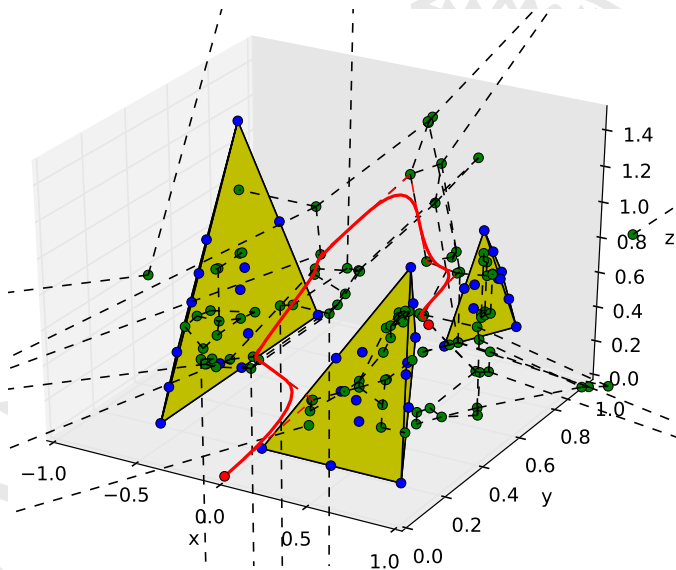
Path creation

```
1 u = endTriplet
2 while u in prev:
3     u = prev[u]
4     path[:0] = [u[1]]
5
6 if path:
7     path[len(path):] = [end]
8     path[:0] = [start]
9
10 return np.array(path)
```

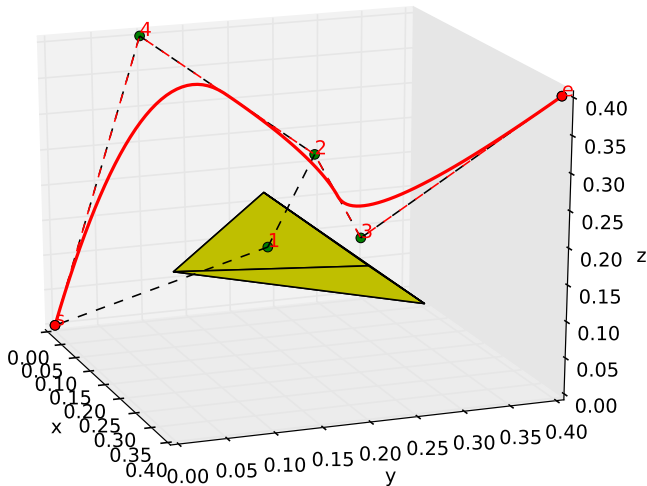
After

we can use the returned **path** as a **control polygon** for a quadratic B-Spline without problems, and construct a smoother path.

Previous example



Clearer example



Complexity considerations

The algorithm is analogous to classical **Dijkstra** applied to a **transformed** graph where:

- ✓ the original graph is **not** directed and weighted
- ✓ the transformed graph is **directed** and weighted, and
 - ▶ if A and B are **neighbouring** and B and C are **neighbouring**, in the original graph
 - ▶ we have two nodes (A, B, C) and (C, B, A) in the transformed graph
 - ▶ a node (A_1, B_1, C_1) is a predecessor of (A_2, B_2, C_2) in the transformed graph if $B_1 = A_2$ and $C_1 = B_2$ in the original graph
 - ▶ and the weight of the arc is the weight of the original from A_1 to $B_1(=A_2)$

Complexity considerations

The algorithm is analogous to classical **Dijkstra** applied to a **transformed** graph where:

- ✓ the original graph is **not** directed and weighted
- ✓ the transformed graph is **directed** and weighted, and
 - ▶ if A and B are **neighbouring** and B and C are **neighbouring**, in the original graph
 - ▶ we have two nodes (A, B, C) and (C, B, A) in the transformed graph
 - ▶ a node (A_1, B_1, C_1) is a predecessor of (A_2, B_2, C_2) in the transformed graph if $B_1 = A_2$ and $C_1 = B_2$ in the original graph
 - ▶ and the weight of the arc is the weight of the original from A_1 to $B_1(=A_2)$

Complexity considerations

The algorithm is analogous to classical **Dijkstra** applied to a **transformed** graph where:

- ✓ the original graph is **not** directed and weighted
- ✓ the transformed graph is **directed** and weighted, and
 - ▶ if A and B are **neighbouring** and B and C are **neighbouring**, in the original graph
 - ▶ we have two nodes (A, B, C) and (C, B, A) in the transformed graph
 - ▶ a node (A_1, B_1, C_1) is a predecessor of (A_2, B_2, C_2) in the transformed graph if $B_1 = A_2$ and $C_1 = B_2$ in the original graph
 - ▶ and the weight of the arc is the weight of the original from A_1 to $B_1(=A_2)$

Complexity considerations

The algorithm is analogous to classical **Dijkstra** applied to a **transformed** graph where:

- ✓ the original graph is **not** directed and weighted
- ✓ the transformed graph is **directed** and weighted, and
 - ▶ if **A** and **B** are **neighbouring** and **B** and **C** are **neighbouring**, in the original graph
 - ▶ we have two nodes **(A, B, C)** and **(C, B, A)** in the transformed graph
 - ▶ a node **(A_1, B_1, C_1)** is a predecessor of **(A_2, B_2, C_2)** in the transformed graph if **$B_1 = A_2$** and **$C_1 = B_2$** in the original graph
 - ▶ and the weight of the arc is the weight of the original from **A_1** to **$B_1 (= A_2)$**

Complexity considerations

The algorithm is analogous to classical **Dijkstra** applied to a **transformed** graph where:

- ✓ the original graph is **not** directed and weighted
- ✓ the transformed graph is **directed** and weighted, and
 - ▶ if **A** and **B** are **neighbouring** and **B** and **C** are **neighbouring**, in the original graph
 - ▶ we have two nodes **(A, B, C)** and **(C, B, A)** in the transformed graph
 - ▶ a node **(A_1, B_1, C_1)** is a predecessor of **(A_2, B_2, C_2)** in the transformed graph if **$B_1 = A_2$** and **$C_1 = B_2$** in the original graph
 - ▶ and the weight of the arc is the weight of the original from **A_1** to **$B_1 (= A_2)$**

Complexity considerations

The algorithm is analogous to classical **Dijkstra** applied to a **transformed** graph where:



- ✓ the original graph is **not** directed and weighted
- ✓ the transformed graph is **directed** and weighted, and
 - ▶ if A and B are **neighbouring** and B and C are **neighbouring**, in the original graph
 - ▶ we have two nodes (A, B, C) and (C, B, A) in the transformed graph
 - ▶ a node (A_1, B_1, C_1) is a predecessor of (A_2, B_2, C_2) in the transformed graph if $B_1 = A_2$ and $C_1 = B_2$ in the original graph
 - ▶ and the weight of the arc is the weight of the original from A_1 to $B_1 (= A_2)$

Complexity considerations



The algorithm is analogous to classical **Dijkstra** applied to a **transformed** graph where:

- ✓ the original graph is **not** directed and weighted
- ✓ the transformed graph is **directed** and weighted, and
 - ▶ if A and B are **neighbouring** and B and C are **neighbouring**, in the original graph
 - ▶ we have two nodes (A, B, C) and (C, B, A) in the transformed graph
 - ▶ a node (A_1, B_1, C_1) is a predecessor of (A_2, B_2, C_2) in the transformed graph if $B_1 = A_2$ and $C_1 = B_2$ in the original graph
 - ▶ and the weight of the arc is the weight of the original from A_1 to $B_1 (= A_2)$

Time complexity (After creation of the original graph)

- ✓ Suppose that the graph has a maximum degree k
 - ▶ each node has maximum k neighbours
- ✓ Each node is the central point of $2 \cdot \binom{k}{2} = k \cdot (k-1)$ triples 
- ▶ $|V_{mod}| = |V_{orig}| \cdot k \cdot (k-1) = \mathcal{O}(k^2 |V_{orig}|)$
- ✓ Each triple is a predecessor of $k-1$ triples 
- ▶ $|E_{mod}| = |V_{mod}| \cdot (k-1) = \mathcal{O}(k |V_{mod}|) = \mathcal{O}(k^3 |V_{orig}|)$
- ✓ Cost of Dijkstra:
 $\mathcal{O}(|E_{mod}| + |V_{mod}| \log |V_{mod}|) = \mathcal{O}(k^3 |V_{orig}| + k^2 |V_{orig}| \log(k |V_{orig}|))$
- ✓ Negligible cost for triples creation: $|V_{orig}| \cdot k \cdot (k-1) = \mathcal{O}(k^2 |V_{orig}|)$

Time complexity (After creation of the original graph)

- ✓ Suppose that the graph has a maximum degree k
 - ▶ each node has maximum k neighbours
- ✓ Each node is the central point of $2 \cdot \binom{k}{2} = k \cdot (k-1)$ triples
- ▶ $|V_{mod}| = |V_{orig}| \cdot k \cdot (k-1) = \mathcal{O}(k^2 |V_{orig}|)$
- ✓ Each triple is a predecessor of $k-1$ triples
- ▶ $|E_{mod}| = |V_{mod}| \cdot (k-1) = \mathcal{O}(k |V_{mod}|) = \mathcal{O}(k^3 |V_{orig}|)$
- ✓ Cost of Dijkstra:
 $\mathcal{O}(|E_{mod}| + |V_{mod}| \log |V_{mod}|) = \mathcal{O}(k^3 |V_{orig}| + k^2 |V_{orig}| \log(k |V_{orig}|))$
- ✓ Negligible cost for triples creation: $|V_{orig}| \cdot k \cdot (k-1) = \mathcal{O}(k^2 |V_{orig}|)$

Time complexity (After creation of the original graph)

- ✓ Suppose that the graph has a maximum degree k
 - ▶ each node has maximum k neighbours
- ✓ Each node is the central point of $2 \cdot \binom{k}{2} = k \cdot (k - 1)$ triples



- ▶ $|V_{mod}| = |V_{orig}| \cdot k \cdot (k - 1) = \mathcal{O}(k^2 |V_{orig}|)$

- ✓ Each triple is a predecessor of $k - 1$ triples

- ▶ $|E_{mod}| = |V_{mod}| \cdot (k - 1) = \mathcal{O}(k |V_{mod}|) = \mathcal{O}(k^3 |V_{orig}|)$



- ✓ Cost of Dijkstra:

$$\mathcal{O}(|E_{mod}| + |V_{mod}| \log |V_{mod}|) = \mathcal{O}(k^3 |V_{orig}| + k^2 |V_{orig}| \log(k |V_{orig}|))$$

- ✓ Negligible cost for triples creation: $|V_{orig}| \cdot k \cdot (k - 1) = \mathcal{O}(k^2 |V_{orig}|)$

Time complexity (After creation of the original graph)

- ✓ Suppose that the graph has a maximum degree k
 - ▶ each node has maximum k neighbours
- ✓ Each node is the central point of $2 \cdot \binom{k}{2} = k \cdot (k - 1)$ triples



- ▶ $|V_{mod}| = |V_{orig}| \cdot k \cdot (k - 1) = \mathcal{O}(k^2 |V_{orig}|)$
- ✓ Each triple is a predecessor of $k - 1$ triples
 - ▶ $|E_{mod}| = |V_{mod}| \cdot (k - 1) = \mathcal{O}(k |V_{mod}|) = \mathcal{O}(k^3 |V_{orig}|)$



- ✓ Cost of Dijkstra:
 $\mathcal{O}(|E_{mod}| + |V_{mod}| \log |V_{mod}|) = \mathcal{O}(k^3 |V_{orig}| + k^2 |V_{orig}| \log(k |V_{orig}|))$
- ✓ Negligible cost for triples creation: $|V_{orig}| \cdot k \cdot (k - 1) = \mathcal{O}(k^2 |V_{orig}|)$

Time complexity (After creation of the original graph)

- ✓ Suppose that the graph has a maximum degree k
 - ▶ each node has maximum k neighbours
- ✓ Each node is the central point of $2 \cdot \binom{k}{2} = k \cdot (k - 1)$ triples
 - ▶ $|V_{mod}| = |V_{orig}| \cdot k \cdot (k - 1) = \mathcal{O}(k^2 |V_{orig}|)$



- ✓ Each triple is a predecessor of $k - 1$ triples



- ▶ $|E_{mod}| = |V_{mod}| \cdot (k - 1) = \mathcal{O}(k |V_{mod}|) = \mathcal{O}(k^3 |V_{orig}|)$
- ✓ Cost of Dijkstra:
 $\mathcal{O}(|E_{mod}| + |V_{mod}| \log |V_{mod}|) = \mathcal{O}(k^3 |V_{orig}| + k^2 |V_{orig}| \log(k |V_{orig}|))$
- ✓ Negligible cost for triples creation: $|V_{orig}| \cdot k \cdot (k - 1) = \mathcal{O}(k^2 |V_{orig}|)$

Time complexity (After creation of the original graph)

- ✓ Suppose that the graph has a maximum degree k
 - ▶ each node has maximum k neighbours
- ✓ Each node is the central point of $2 \cdot \binom{k}{2} = k \cdot (k - 1)$ triples



- ▶ $|V_{mod}| = |V_{orig}| \cdot k \cdot (k - 1) = \mathcal{O}(k^2 |V_{orig}|)$

- ✓ Each triple is a predecessor of $k - 1$ triples



- ▶ $|E_{mod}| = |V_{mod}| \cdot (k - 1) = \mathcal{O}(k |V_{mod}|) = \mathcal{O}(k^3 |V_{orig}|)$

- ✓ Cost of Dijkstra:

$$\mathcal{O}(|E_{mod}| + |V_{mod}| \log |V_{mod}|) = \mathcal{O}(k^3 |V_{orig}| + k^2 |V_{orig}| \log(k |V_{orig}|))$$

- ✓ Negligible cost for triples creation: $|V_{orig}| \cdot k \cdot (k - 1) = \mathcal{O}(k^2 |V_{orig}|)$

Time complexity (After creation of the original graph)

- ✓ Suppose that the graph has a maximum degree k
 - ▶ each node has maximum k neighbours
- ✓ Each node is the central point of $2 \cdot \binom{k}{2} = k \cdot (k - 1)$ triples



- ▶ $|V_{mod}| = |V_{orig}| \cdot k \cdot (k - 1) = \mathcal{O}(k^2 |V_{orig}|)$

- ✓ Each triple is a predecessor of $k - 1$ triples



- ▶ $|E_{mod}| = |V_{mod}| \cdot (k - 1) = \mathcal{O}(k |V_{mod}|) = \mathcal{O}(k^3 |V_{orig}|)$

- ✓ Cost of Dijkstra:

$$\mathcal{O}(|E_{mod}| + |V_{mod}| \log |V_{mod}|) = \mathcal{O}(k^3 |V_{orig}| + k^2 |V_{orig}| \log(k |V_{orig}|))$$

- ✓ Negligible cost for triples creation: $|V_{orig}| \cdot k \cdot (k - 1) = \mathcal{O}(k^2 |V_{orig}|)$

Time complexity (After creation of the original graph)

- ✓ Suppose that the graph has a maximum degree k
 - ▶ each node has maximum k neighbours
- ✓ Each node is the central point of $2 \cdot \binom{k}{2} = k \cdot (k - 1)$ triples



- ▶ $|V_{mod}| = |V_{orig}| \cdot k \cdot (k - 1) = \mathcal{O}(k^2 |V_{orig}|)$

- ✓ Each triple is a predecessor of $k - 1$ triples



- ▶ $|E_{mod}| = |V_{mod}| \cdot (k - 1) = \mathcal{O}(k |V_{mod}|) = \mathcal{O}(k^3 |V_{orig}|)$

- ✓ Cost of Dijkstra:

$$\mathcal{O}(|E_{mod}| + |V_{mod}| \log |V_{mod}|) = \mathcal{O}(k^3 |V_{orig}| + k^2 |V_{orig}| \log(k |V_{orig}|))$$

- ✓ Negligible cost for triples creation: $|V_{orig}| \cdot k \cdot (k - 1) = \mathcal{O}(k^2 |V_{orig}|)$

Special cases

✓ $\mathcal{O}(k^3|V_{orig}| + k^2|V_{orig}|\log(k|V_{orig}|))$

If k is constant (don't grow with $|V_{orig}|$)

- ✓ Total cost: $\mathcal{O}(|V_{orig}|\log|V_{orig}|)$
- ✓ same cost of Dijkstra in a lattice

If the graph is a clique

- ✓ $k = |V_{orig}| - 1$
- ✓ Total cost: $\mathcal{O}(|V_{orig}|^4)$

Special cases

✓ $\mathcal{O}(k^3|V_{orig}| + k^2|V_{orig}|\log(k|V_{orig}|))$

If k is constant (don't grow with $|V_{orig}|$)

- ✓ Total cost: $\mathcal{O}(|V_{orig}|\log|V_{orig}|)$
- ✓ same cost of Dijkstra in a lattice

If the graph is a clique

- ✓ $k = |V_{orig}| - 1$
- ✓ Total cost: $\mathcal{O}(|V_{orig}|^4)$

Special cases

✓ $\mathcal{O}(k^3|V_{orig}| + k^2|V_{orig}|\log(k|V_{orig}|))$

If k is constant (don't grow with $|V_{orig}|$)

- ✓ Total cost: $\mathcal{O}(|V_{orig}|\log|V_{orig}|)$
- ✓ same cost of Dijkstra in a lattice

If the graph is a clique

- ✓ $k = |V_{orig}| - 1$
- ✓ Total cost: $\mathcal{O}(|V_{orig}|^4)$

Special cases

✓ $\mathcal{O}(k^3|V_{orig}| + k^2|V_{orig}|\log(k|V_{orig}|))$

If k is constant (don't grow with $|V_{orig}|$)

- ✓ Total cost: $\mathcal{O}(|V_{orig}|\log|V_{orig}|)$
- ✓ same cost of Dijkstra in a lattice

If the graph is a clique

- ✓ $k = |V_{orig}| - 1$
- ✓ Total cost: $\mathcal{O}(|V_{orig}|^4)$

Special cases

✓ $\mathcal{O}(k^3|V_{orig}| + k^2|V_{orig}|\log(k|V_{orig}|))$

If k is constant (don't grow with $|V_{orig}|$)

- ✓ Total cost: $\mathcal{O}(|V_{orig}|\log|V_{orig}|)$
- ✓ same cost of Dijkstra in a lattice

If the graph is a clique

- ✓ $k = |V_{orig}| - 1$
- ✓ Total cost: $\mathcal{O}(|V_{orig}|^4)$

The End.

Questions? Thank you!

The End.

Questions? Thank you!

The End.

Questions? Thank you!