# Trijkstra

## A Dijkstra algorithm application to path planning

Stefano Martina
stefano.martina@stud.unifi.it

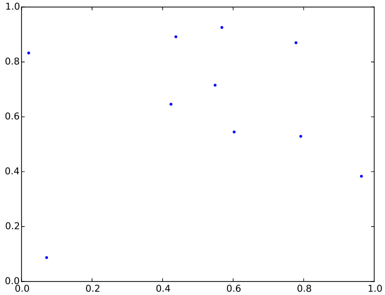
UNIVERSITÀ DEGLI STUDI FIRENZE

4 December 2015

# Voronoi diagrams

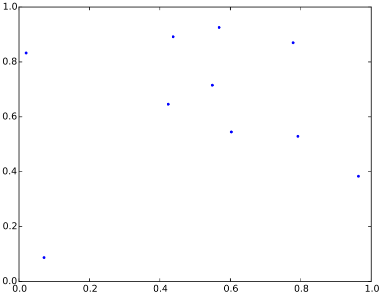
**Input:** A set of points in plane (or space) called sites

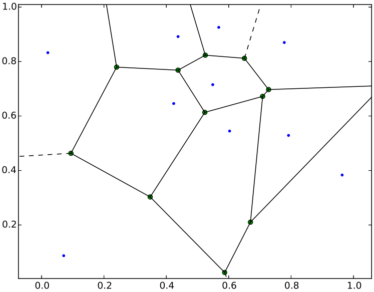
**Output:** A partition of the plane (or space) such that each point of a region is nearer to a certain site respect to the others



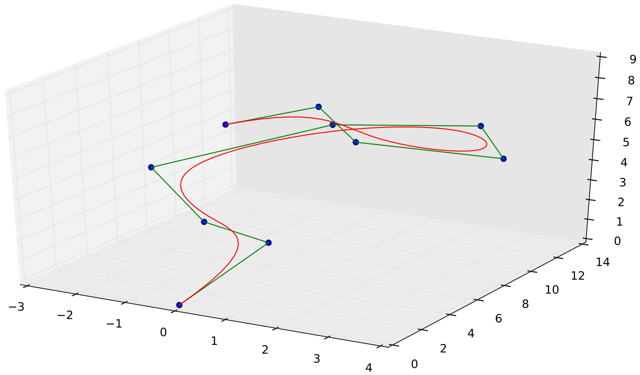
**Input:** A set of points in plane (or space) called sites

**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



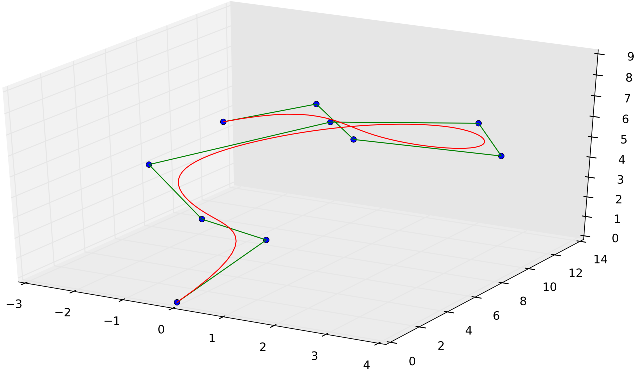



# B-spline



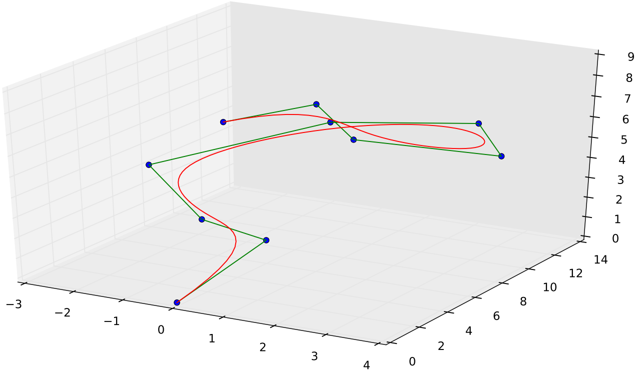

- ✓ parametric curves
- ✓ follow the shape of a control poligon
- ✓ can interpolate the extremes of the control polygon

# B-spline



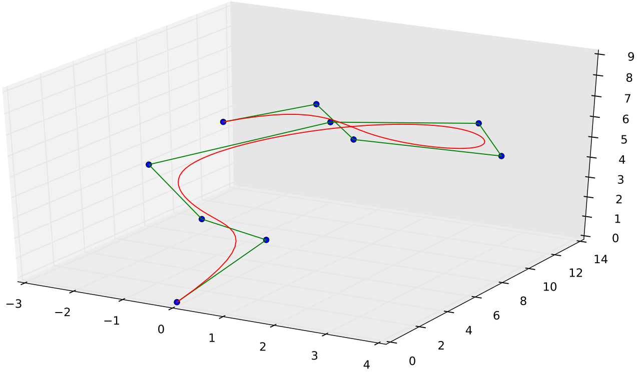

- ✓ parametric curves
- ✓ follow the shape of a control poligon
- ✓ can interpolate the extremes of the control polygon

# 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

# 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.

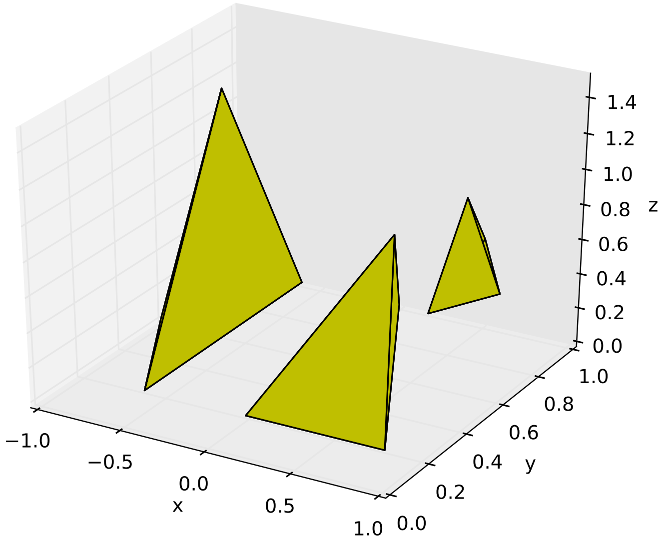
# Background

## Main problem

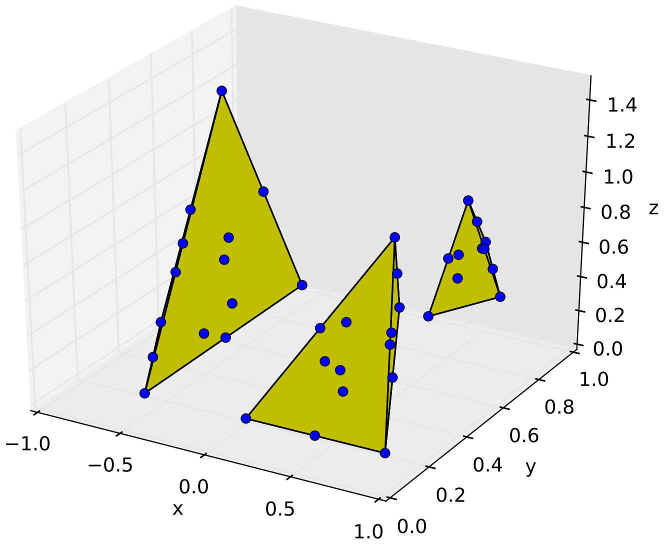
Path planning from a start point to an end point in 3D space with obstacles using Voronoi diagrams.

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.

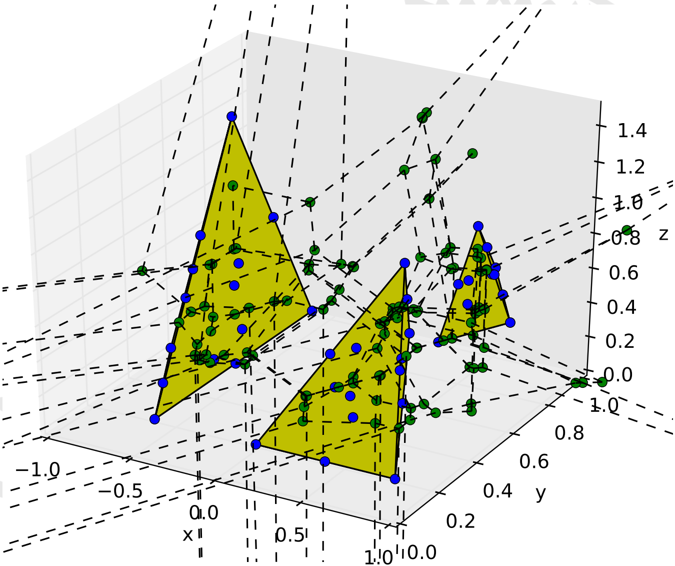
# Example



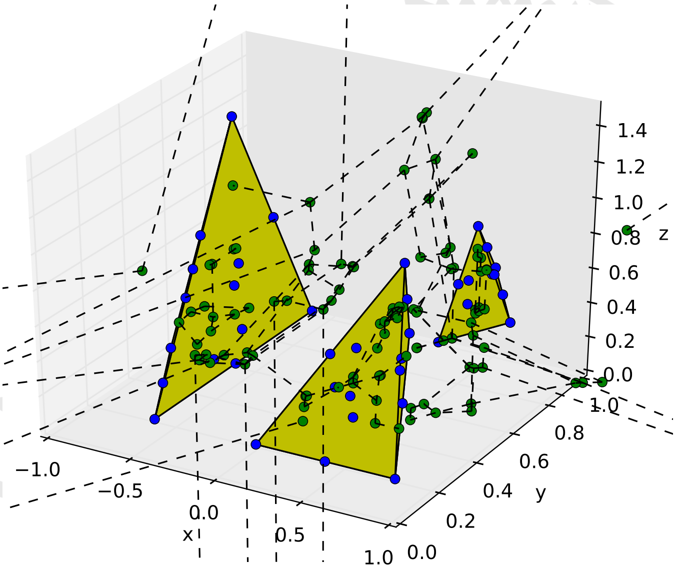
# Example



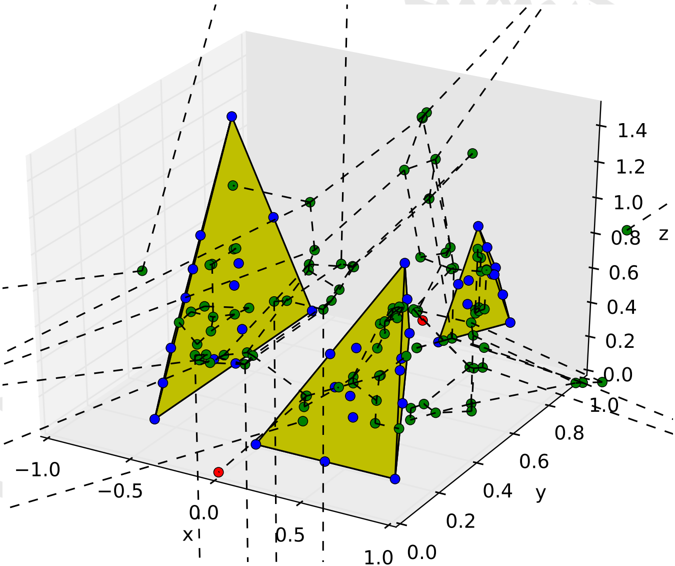
# Example



# 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

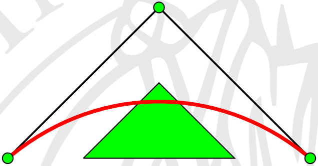
# Problem

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

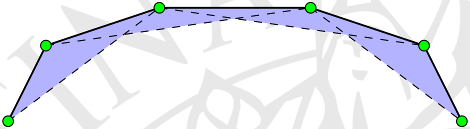


**Idea**

- ✓ 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



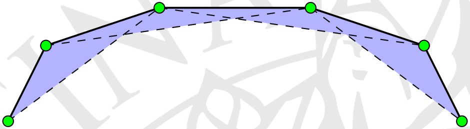
**Idea**

✓ 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



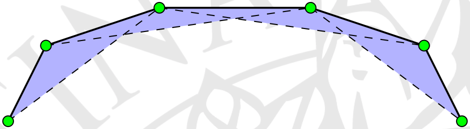
**Idea**

✓ 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 $\infty$ 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 $\infty$
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 $\infty$ 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 $\infty$
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 $\infty$ 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 $\infty$
7. the shortest path (with free triangular convex hull) can be obtained following the previous pointer from the ending triple, and deconstructing the triples

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 $\infty$ 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 $\infty$
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 $\infty$ 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 $\infty$
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 $\infty$ 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 $\infty$
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 $\infty$ 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 $\infty$
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 $\infty$ 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 $\infty$
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 $\infty$ 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 $\infty$
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 $\infty$ 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 $\infty$
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 $\infty$ 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 $\infty$
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 $\infty$ 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 $\infty$
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

```python
def _trijkstra(self, startA, endA):
  start = tuple(startA)
  end = tuple(endA)
  endTriplet = (end,end,end) #special triplet for termination
  inf = float("inf")
  path = []
  Q = priorityQueue.PQueue()
  dist = {}
  prev = {}
  hits = []
```

```python
    for node0 in self._graph.nodes():
      for node1 in self._graph.neighbors(node0):
        for node2 in filter(lambda node: node!=node0, self._graph.neighbors(node1)):
          triplet = (node0,node1,node2)
          if not triplet[::-1] in hits:
            if not self._triangleIntersectPolyhedrons(np.array(node0), np.array(node1),
                 ↪ np.array(node2)):
              if node0 != start:
                dist[triplet] = inf
                Q.add(triplet, inf)
              else:
                dist[triplet] = 0
                Q.add(triplet, 0)
            else:
              hits[:0] = [triplet]

    dist[endTriplet] = inf
    Q.add(endTriplet, inf)
```

# Main loop

```
while True:
  u = Q.pop()

  if u == endTriplet or dist[u] == inf:
    break

  for v in Q.filterGet(lambda tri: u[1] == tri[0] and u[2] == tri[1]):
    tmpDist = dist[u] + self._graph[u[0]][u[1]]['weight']
    if tmpDist < dist[v]:
      dist[v] = tmpDist
      prev[v] = u
      Q.add(v, tmpDist)

  if u[2] == end:
    tmpDist = dist[u] + self._graph[u[0]][u[1]]['weight'] + self._graph[u[1]][u[2]]['weight']
    if tmpDist < dist[endTriplet]:
      dist[endTriplet] = tmpDist
      prev[endTriplet] = u
      Q.add(v, tmpDist)
```

# Path creation

```
u = endTriplet
while u in prev:
  u = prev[u]
  path[:0] = [u[1]]

if path:
  path[len(path):] = [end]
  path[:0] = [start]

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.

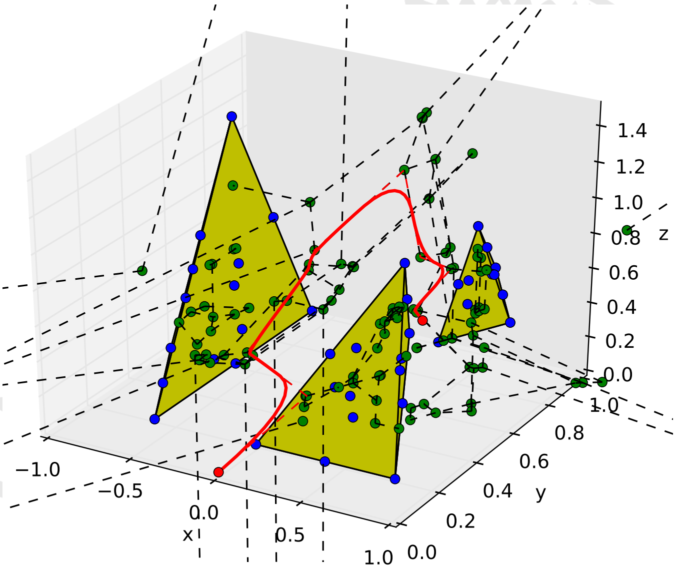
# Path creation

```
u = endTriplet
while u in prev:
  u = prev[u]
  path[:0] = [u[1]]

if path:
  path[len(path):] = [end]
  path[:0] = [start]

return np.array(path)
```

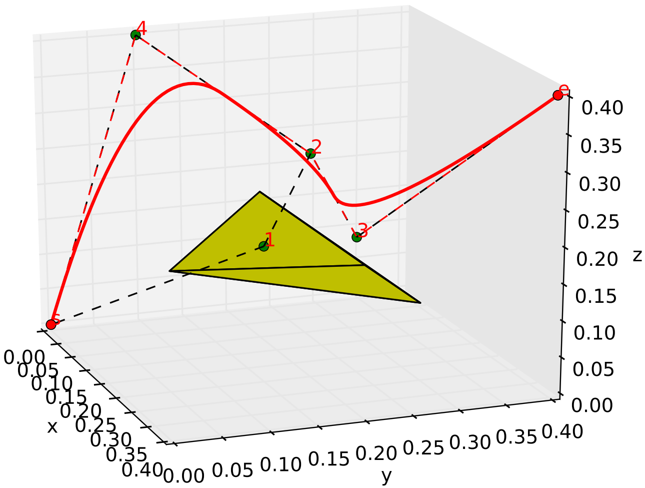
## After

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!*