BILKENT UNIVERSITY
ENGINEERING FACULTY
DEPARTMENT OF COMPUTER ENGINEERING

CS 478

Project Description
Delaunay Triangulation In 2D

Gürkan Gür
21602813

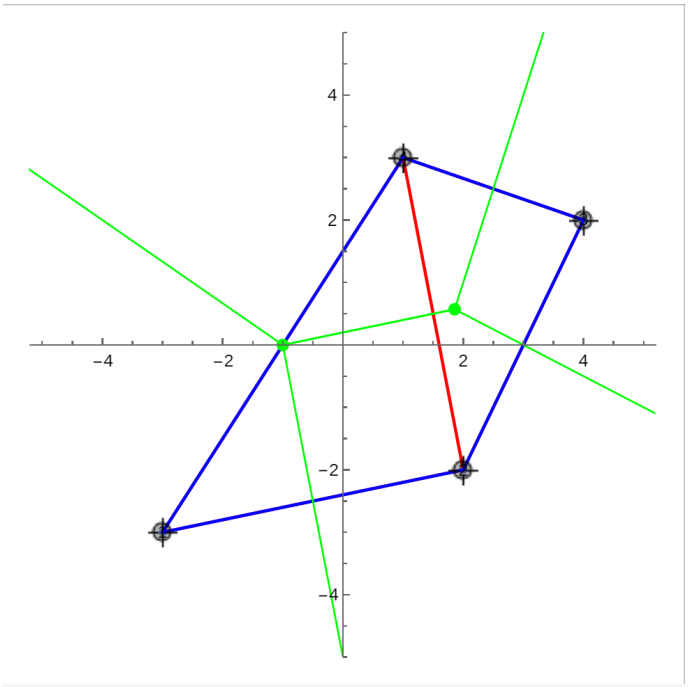
Table of Contents

Survey

A triangulation is a subdivision of an area (volume) into triangles (tetrahedrons). The Delaunay triangulation has the property that the circumcircle (circumsphere) of every triangle (tetrahedron) does not contain any points of the triangulation.[1] Every circumcircle of a triangle is an empty circle (Okabe et al. 1992, p. 94).

Delaunay triangulations, convex hulls and Voronoi diagrams have a connection between them. In the context of 2 dimensions, “The lifting map λ : R^2 → R^3 is defined by λ(x1, x2) = (x1, x2, x2 1 + x 2 2); Λ = λ(R 2) is a paraboloid of revolution about the vertical axis.”[2] If the lower faces(a face is lower if it has a supporting plane with inward normal having positive vertical coordinate) of a convex hull of the lifted sites orthogonally projected into R^2, Delaunay triangulation of the same point set is obtained.[2]

Connection between a delaunay triangulation and a Voronoi diagram is described in [2] as “...suppose that triangle λ(s)λ(t)λ(u) is a lower facet of H, and that plane P passes through λ(s)λ(t)λ(u). The intersection of P with Λ is an ellipse that projects orthogonally to a circle in R^2. Since all other lifted sites are above the plane, all other unlifted sites are outside the circle, and stu is a Delaunay triangle. The opposite direction, that a Delaunay triangle is a lower facet, is similar. For Voronoi diagrams, assign to each site s = (s1, s2) the plane Ps = {(x1, x2, x3) : x3 = −2x1s1 + s 2 1 − 2x2s2 + s 2 2 }. Let I be the intersection of the lower halfspaces of the plane's Ps. The Voronoi diagram is exactly the orthogonal projection into R^2 of the upper faces of I.”



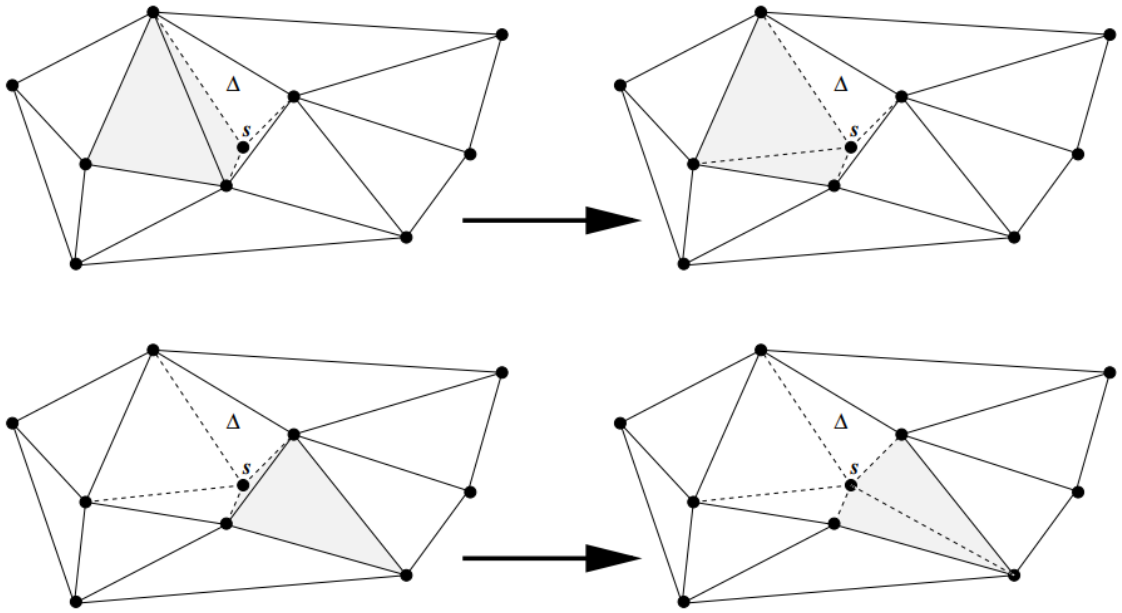
An illustration of Delaunay triangulation(blue and red), convex hull(blue) and Voronoi diagram(green).[3]

Algorithms and Data Structures

Convex hull algorithms(the divide-and-conquer, incremental, and gift-wrapping algorithms) can be used to compute Delaunay triangulations.
Randomized Incremental Algorithm and Plane sweep algorithms will be used in this project.

Randomized Incremental Algorithm

A randomized incremental algorithm adds vertices one by one. After each addition, triangulation is checked and updated. The update consists of discovering all Delaunay faces.[2]



An illustration of the update phase. s is the new added vertex.[5]

Initial Step

On an empty plane, think of 3 artificial points outside of all possible points. Add a random point from the point set.

Method

Find the triangle that contains the newly added point. Connect the vertices of the triangle with the newly added point.
Perform flips to create Delaunay triangulation. At first glance, it may seem like we need to search all the triangles and perform flips on them. However, if we sign the edges that need to be changed as “bad”, and that can stay the same as “good”;
“(a) In every flip, the convex quadrilateral Q in which the flip happens has exactly two edges incident to ps(new point), and the flip generates a new edge incident to ps.
(b) Only the two edges of Q that are not incident to ps can become bad after the flip.”
Considering these 2 rules, a queue of potentially bad edges can be maintained.
A good edge will be removed from the queue, and a bad edge will be flipped and replaced according to (b) with two new edges in the queue.

Data Structures

Point Class:

```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
    this.z = 0.0;
    this.w = 1.0;
  }
  equal(p2) {

    if(this.x == p2.x && this.y==p2.y)
    return true;
    else
    return false;
  }

}
```

Edge Class:

```
class Edge{
    constructor(p1,p2){
        this.p1 = p1;
        this.p2 = p2;
        this.t1 = null;
        this.t2 = null;
    }
    equal(edge2){
        if((this.p1.equal(edge2.p1) && this.p2.equal(edge2.p2)) ||
(this.p2.equal(edge2.p1) && this.p1.equal(edge2.p2)))
        return true;
    }
    setT1(tt){
        this.t1 = tt;
    }
    setT2(tt){
        this.t2 = tt;
    }
    containsPoint(p){
        if(this.p1.equal(p) || this.p2.equal(p))
        return true;
        else
        return false;
    }
    intersects(edge){
        if(this.p1.equal(edge.p1) || this.p1.equal(edge.p2) ||
this.p2.equal(edge.p1) || this.p2.equal(edge.p2))
        return true;
        else
        return false;
    }
    getOtherTriangle(tt){
        if(this.t1!=null && tt.equal(this.t1))
        return this.t2;
        else if(this.t2!=null && tt.equal(this.t2))
        return this.t1;

        return null;
    }
}
```

Triangle Class:

```
class Triangle {
    constructor(edge1, edge2, edge3, p1, p2, p3){
        this.edge1 = edge1;
        this.edge2 = edge2;
        this.edge3 = edge3;
        this.p1 = p1;
        this.p2 = p2;
        this.p3 = p3;
        this.adT1=null;
        this.adT2=null;
        this.adT3=null;
        this.activity = true;
        this.index=null;
    }
    equal(triangle2){
        if(this.p1.equal(triangle2.p1)){
            if(this.p2.equal(triangle2.p2)){
                if(this.p3.equal(triangle2.p3)){
                return true;
            }
            else{
                return false;
            }
            }
            else if(this.p2.equal(triangle2.p3)){
                if(this.p3.equal(triangle2.p2)){
                return true;
            }
            else{
                return false;
            }
            }
            else
            return false;
        }
        else if(this.p1.equal(triangle2.p2)){

            if(this.p2.equal(triangle2.p3)){
                if(this.p3.equal(triangle2.p1)){
                return true;
            }
            else{
```

```
                return false;
            }
            }
            else if(this.p2.equal(triangle2.p1)){
                if(this.p3.equal(triangle2.p3)){
                return true;
            }
            else{
                return false;
            }
            }
            else{
                return false;
            }
        }
        else if(this.p1.equal(triangle2.p3)){
            if(this.p2.equal(triangle2.p1)){
                if(this.p3.equal(triangle2.p2)){
                return true;
            }
            else{
                return false;
            }
            }
            else if(this.p2.equal(triangle2.p2)){
                if(this.p3.equal(triangle2.p1)){
                return true;
            }
            else{
                return false;
            }
            }
            else{
                return false;
            }
        }
        else{
            return false;
        }
    }
    setEdges(e1,e2,e3){
        this.edge1 = e1;
        this.edge2 = e2;
```

```
        this.edge3 = e3;
    }
    activate(){
        this.activity = true;
    }
    deactivate(){
        this.activity = false;
    }
    getFrontPoint(edge){
        if(edge.equal(this.edge1)){
            if(this.edge2.p1.equal(edge.p1) ||
this.edge2.p1.equal(edge.p2)){
                return this.edge2.p2
            }
            else if(this.edge2.p2.equal(edge.p1) ||
this.edge2.p2.equal(edge.p2)){
                return this.edge2.p1
            }
        }
        else if(edge.equal(this.edge2)){
            if(this.edge1.p1.equal(edge.p1) ||
this.edge1.p1.equal(edge.p2)){
                return this.edge1.p2
            }
            else if(this.edge1.p2.equal(edge.p1) ||
this.edge1.p2.equal(edge.p2)){
                return this.edge1.p1
            }
        }
        else if(edge.equal(this.edge3)){
            if(this.edge1.p1.equal(edge.p1) ||
this.edge1.p1.equal(edge.p2)){
                return this.edge1.p2
            }
            else if(this.edge1.p2.equal(edge.p1) ||
this.edge1.p2.equal(edge.p2)){
                return this.edge1.p1
            }
        }

    }
```

Creating Random Points

```
for (i = 0; i < n; i += 1) {
        var p = new Point(gaussianRand(),gaussianRand());
        arr2.push(p);
        arr.push(p.x);
        arr.push(p.y);
        arr.push(0.0);
        arr.push(1.0);
    }
```

Random points are created using Gaussian Distribution to set an initial triangle easily.
After the creation of the point set, the draw buffer is filled.

```
gl.bindBuffer(gl.ARRAY_BUFFER, vertex_buffer);

        gl.bufferData(gl.ARRAY_BUFFER, 2000000, gl.STATIC_DRAW);
        gl.bufferSubData(gl.ARRAY_BUFFER, 0, new Float32Array(arr));
        gl.vertexAttribPointer(shader.vertex_attrib, 4, gl.FLOAT,
false, 0, 0);
```

Set transform(draw method) is called, and drawing of the points are done.

```
function set_transform ( k, tx, ty ) {

        var matrix = new Float32Array([
            k, 0, 0, 0,
            0, k, 0, 0,
            0, 0, 1, 0,
            2*tx/width, -2*ty/height, 0, 1
        ]);

        gl.uniformMatrix4fv( shader.matrix_uniform, false, matrix );
        gl.clear( gl.COLOR_BUFFER_BIT );
        gl.drawArrays( gl.points, 0, n );
  var h;

  for(h=n; h<n+draw_size; h+=3){
    gl.drawArrays( gl.LINE_LOOP, h, 3 );
  }



    }
```

Triangulation

In the first phase of the triangulation, an initial triangle guaranteed to contain all the points is created.

```
var p1 = new Point(1.0,1.0);
  var p2 = new Point(-1.0,1.0);
  var p3 = new Point(0.0,-1.0);

  var e1 = new Edge(p1,p2);
  var e2 = new Edge(p2,p3);
  var e3 = new Edge(p3,p1);

  var t1 = new Triangle(e1,e2,e3,p1,p2,p3);
```

After that, a point from the point set is selected

```
for(k=0; k<n; k+=1){
    pr = arr2[k];
```

The triangle containing the point is found:

```
for(s=0; s<t_size; s+=1){
        if(T[s].activity==true && InsideTriangle(pr,T[s]))
            {
            //Triangle has been found
            tri_index=s;
            con = 1;
            console.log("found");
            break;
            }
        else{
            con=0;
        }
    }
```

The founded triangle divided by 3, creating 3 new triangles:

```
T[tri_index].deactivate();
        var e11 = new Edge(pr, T[tri_index].p1);
        var e12 = new Edge(pr,T[tri_index].p2);
        var e13 = new Edge(pr,T[tri_index].p3);

        var ec1 =
T[tri_index].getEdge(T[tri_index].p1,T[tri_index].p2);
        var tt1 = new Triangle(e11, e12, ec1, pr,ec1.p1,ec1.p2);
```

```
        var ec2 =
T[tri_index].getEdge(T[tri_index].p1,T[tri_index].p3);
        var tt2 = new Triangle(e11, e13, ec2, pr,ec2.p1,ec2.p2);

        var ec3 =
T[tri_index].getEdge(T[tri_index].p2,T[tri_index].p3);
        var tt3 = new Triangle(e12, e13, ec3, pr,ec3.p1,ec3.p2);
```

Legalize edge procedure is called for the newly added point and its front edges:

```
  LEGALIZEEDGE(pr,ec1,t_size-3);
        LEGALIZEEDGE(pr,ec2,t_size-2);
        LEGALIZEEDGE(pr,ec3,t_size-1);
```

Legalize Edge procedure takes a point, an edge, and the current triangle.
It finds the adjacent triangle of the given edge without searching, because it is stored.
Edge's front point in the adjacent triangle is selected and tested if it is in the circle created from the edge-pr triangle.
If it is, flip happens.

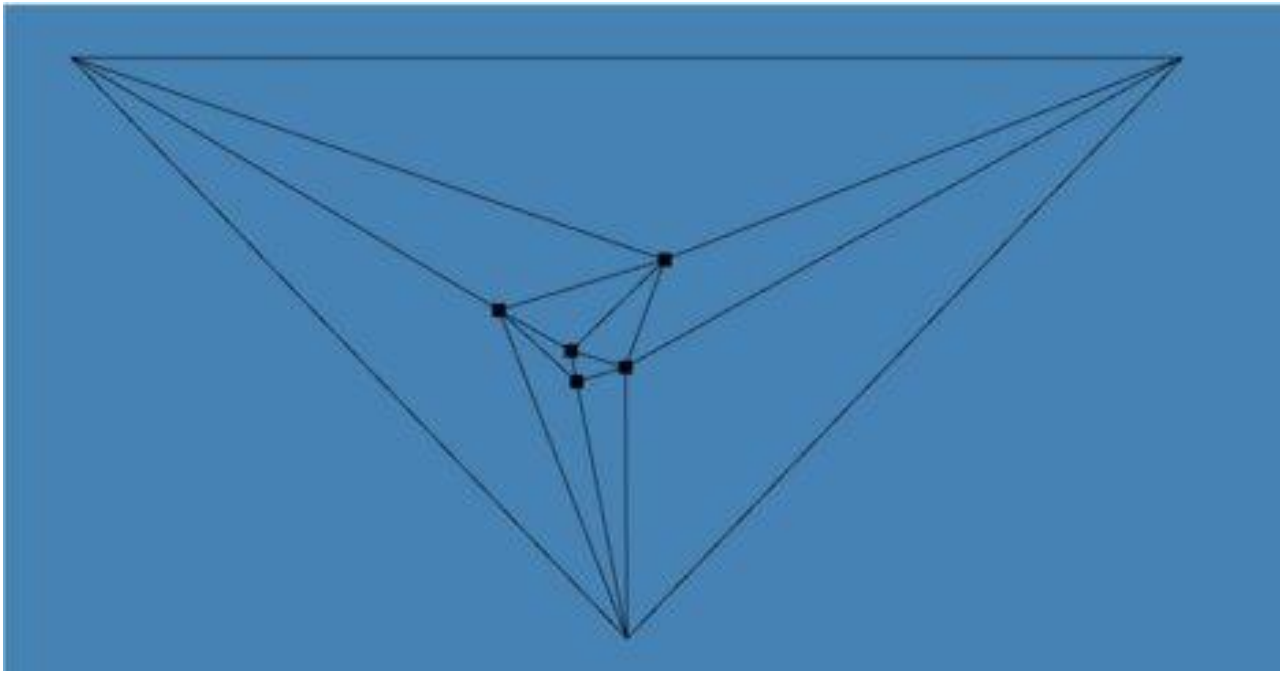
```
function LEGALIZEEDGE(pr, edge, index){

var cent;
var r;
    //Find the triangle adjacent with edge
    var i3;
    var adjTriangle = edge.getOtherTriangle(T[index]);
    if(adjTriangle!=null && adjTriangle.activity==true){
            //T[i3] is the triangle containing edge
            var pn = adjTriangle.getFrontPoint(edge)
                //The triangle is not pr-edge

                cent = createCircle(pr, edge.p1, edge.p2);
                r = Math.sqrt((cent[0]-pr.x) * (cent[0]-pr.x) +
(cent[1]-pr.y) * (cent[1]-pr.y));
                if((pn.x - cent[0])*(pn.x - cent[0]) + (pn.y -
cent[1])*(pn.y - cent[1]) < r*r){
                    //illegal edge
                    //Delete pr pi pj triangle
                    //Delete pi pn pj triangle
                    console.log(adjTriangle);
                    T[index].activity = false;
                    T[adjTriangle.index].activity = false;
                    //Create pr pi Ti triangle
                    var e1 = T[index].getEdge(pr,edge.p1);
                    var e2 = T[index].getEdge(pr,edge.p2);
```

```
                    var e12 = T[adjTriangle.index].getEdge(pn,edge.p1);
                    var e22 = T[adjTriangle.index].getEdge(pn,edge.p2);

                    var newE = new Edge(pn,pr);

                    var newT1 = new
Triangle(e1,e12,newE,pr,pn,edge.p1);
                    var newT2 = new
Triangle(e2,e22,newE,pr,pn,edge.p2);

                    newE.setT1(newT1);
                    newE.setT2(newT2);
                    e1.setT1(newT1);
                    e1.setT2(e1.getOtherTriangle(T[index]));

                    e2.setT1(newT1);
                    e2.setT2(e2.getOtherTriangle(T[index]));

                    e12.setT1(newT2);

e12.setT2(e12.getOtherTriangle(T[adjTriangle.index]));

                    e22.setT1(newT2);

e22.setT2(e22.getOtherTriangle(T[adjTriangle.index]));

                    t_size+=2;
                    newT1.index=t_size-2;
                    newT2.index=t_size-1;

                    newT1.setEdges(e1,e12,newE);
                    newT2.setEdges(e2,e22,newE);

                    T.push(newT1);
                    T.push(newT2);
                    LEGALIZEEDGE(pr,e12,t_size-2);
                    LEGALIZEEDGE(pr,e22,t_size-1);
                }

    }
}
```

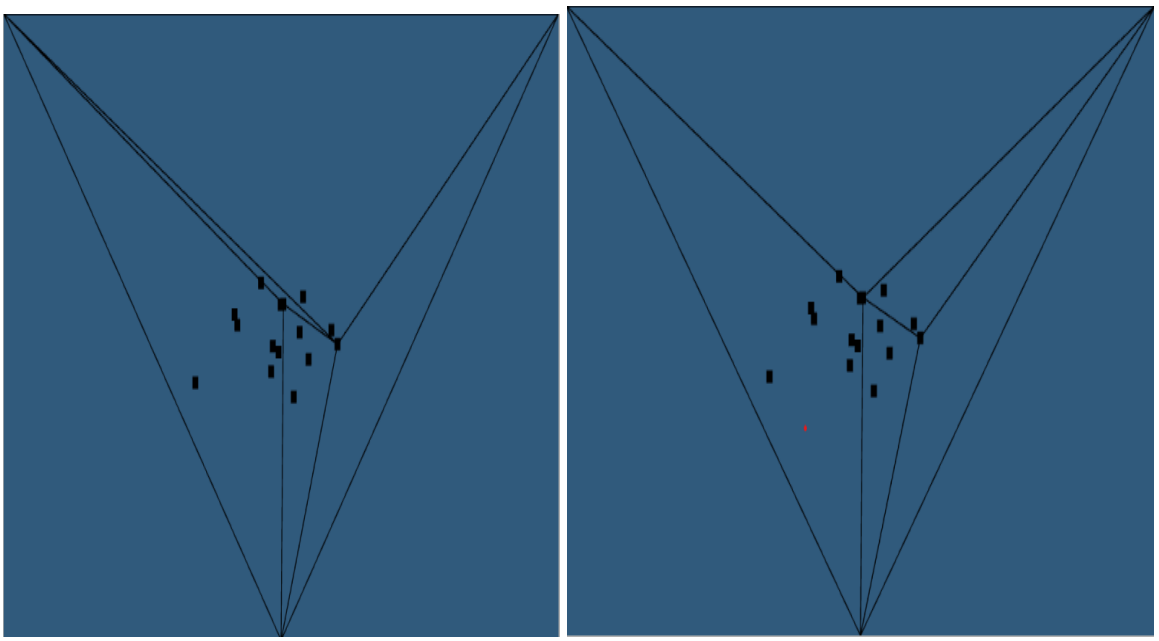
Notes About the Implementation

You can zoom in, zoom out with the mouse wheel.
You can travel by clicking on the plane and moving the mouse.
Initial triangle's edges are not deleted, simply ignore them.
You can select the point count.
You can insert points one by one.
The case of a point on an edge is not considered. It can be thought as a float point can not be on an edge in %100 precision.

Some Screenshots



Triangulation of a point set with size=5



A flip is happening for a random point set

Plane Sweep Algorithm

A sweep-line approach.

2 main data structures:

- frontier: ordered list of sites.
- Event queue: determines sweep line moves with 2 types, site event and circle event.

Site event happens when the sweep line reaches a site.

Circle event happens when the sweep line reaches the top of 3 consecutive vertices.

The algorithm updates the sweep-line data structure and event queue when the sweep-line reaches an event point and discovers a Delaunay triangle when it passes through the circle event.

Sort the points on an array according to ascending y coordinates. Let AS be the resulting array.

Variables
Q //A fifo queue
S
Count //Point count

Initialize first 2 elements of the array
Initialize S = AS[0];
Q.push(S);

```
S = AS[1]
Q.push(S);
```

Main Loop

```
int i = 2 //A loop counter
int qp //Queue index
While( S != AS[count-1]):
        S = AS[i]
        Q.push(S)
        Triangulate(&Q,S);
        i=i+1
```

References

[1] *Delaunay triangulation in two and three dimensions*. Delaunay triangulations. (n.d.). Retrieved March 26, 2022, from http://www.ae.metu.edu.tr/tuncer/ae546/prj/delaunay/?fbclid=IwAR0PC6VdEKvX6eDyHVbyczRgNs4Z0cl9qvMm3XFUEovzc6atiXW5-xZ2QwU

[2] *27 Voronoidiagrams Anddelaunaytriangulations*. (n.d.). Retrieved March 26, 2022, from https://www.csun.edu/~ctoth/Handbook/chap27.pdf

[3] *Wolfram Demonstrations Project*. Convex Hull and Delaunay Triangulation. (n.d.). Retrieved March 26, 2022, from https://demonstrations.wolfram.com/ConvexHullAndDelaunayTriangulation/

[4] *Delaunay triangulations: Properties, algorithms, and ...* (n.d.). Retrieved March 26, 2022, from https://members.loria.fr/MTeillaud/collab/Astonishing/2017_workshop_slides/Olivier_Devillers.pdf

[5] Theory of combinatorial algorithms, Institute of Theoretical Computer Science, Department of Computer Science, ETH Zürich. (n.d.). Retrieved March 26, 2022, from https://www.ti.inf.ethz.ch/ew/Lehre/CG13/