Computational geometry and optimization.

Alexey A. Tuzhilin

Moscow State University
Russia

Computational geometry is devoted to the study of algorithms which can be stated in terms of geometry.

The main branches of computational geometry are
- *combinatorial computational geometry*, which deals with geometric objects as discrete entities (mainly with points, line segments, polygons, polyhedra, etc), and
- *numerical computational geometry*, which deals primarily with representing real-world objects (mainly curve and surface) in forms suitable for computer computations.

We'll consider only some examples of combinatorial computational geometry.

We use Wolfram Mathematica to illustrate

■ Short Introduction to Wolfram Mathematica

Type **command** and then press **Shift+Enter** to tell *Mathematica* to evaluate your input.

```
2 + 2
```

```
4
```

```
(1 + 2. - 5.77 * 3.8) / ((3.2 - 0.1)^(1 / 2) + 6^2)
```

```
-0.501209
```

Note the square brackets from the right side of the text typed

They separates the blocks of commads

The main objects of Mathematica are *Atomic elements* and *Expressions*.

Atomic elements

Numbers

- Integer

```
-13
```

```
-13
```

- Real

```
3.1415
```

```
3.1415
```

- Rational

```
3 / 7
```

$$\frac{3}{7}$$

- Complex

```
4 + 5.1 𝕚
```

```
4 + 5.1 𝕚
```

(to type 𝕚, use palettes)

Symbols

```
x
abc324
```

```
x
```

```
abc324
```

Strings

```
"This is a string"
```

```
This is a string
```

Expressions

They have the form **f[x, y, ...]**

f is called the *head*, x, y, ... are called *arguments* (they can be expressions themselves)

```
Factor[x^9 - 1]
```

$$(-1+x)\left(1+x+x^2\right)\left(1+x^3+x^6\right)$$

Remember: Names of built-in functions and symbols have their first letters capitalized.

```
N[Pi, 100]
```

```
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803
 48825342117068
```

All is expression

```
(1 + 2. - 5.77 * 3.8) / ((3.2 - 0.1)^(1 / 2) + 6^2)
```

is

```
Times[Plus[1, 2.`, Times[-1, Times[5.77`, 3.8`]]],
 Power[Plus[Power[Plus[3.2`, -0.1`], Times[1, Power[2, -1]]], Power[6, 2]], -1]]
```

The main job of Mathematica is to *evaluate expressions* (Shift + Enter)

■ Some examples

- algebraic manipulations

```
Expand[(x + y)^6]
```

$$x^6 + 6\,x^5\,y + 15\,x^4\,y^2 + 20\,x^3\,y^3 + 15\,x^2\,y^4 + 6\,x\,y^5 + y^6$$

- differentiation

```
D[x Sin[x], x]
```

```
x Cos[x] + Sin[x]
```

- integration

```
∫ x Sin[x] dx
```

```
-x Cos[x] + Sin[x]
```

- equations solution

```
Solve[x^3 + 3 x - 4 == 0, x]
```

$$\left\{\{x \to 1\}, \left\{x \to \frac{1}{2}\left(-1 - i\sqrt{15}\right)\right\}, \left\{x \to \frac{1}{2}\left(-1 + i\sqrt{15}\right)\right\}\right\}$$

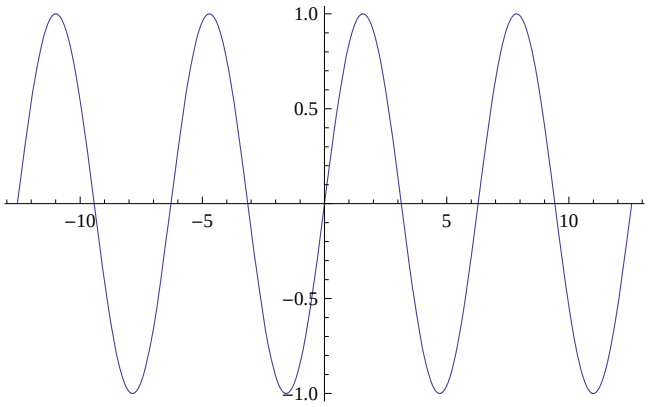
- differential equations solution

```
DSolve[x''[t] - x'[t] + e^t (t^2 + 1) == 0, x[t], t]
```

$$\left\{\left\{x[t] \to -\frac{1}{3}e^t\left(9\,t - 3\,t^2 + t^3 - 3\,(3 + C[1])\right) + C[2]\right\}\right\}$$

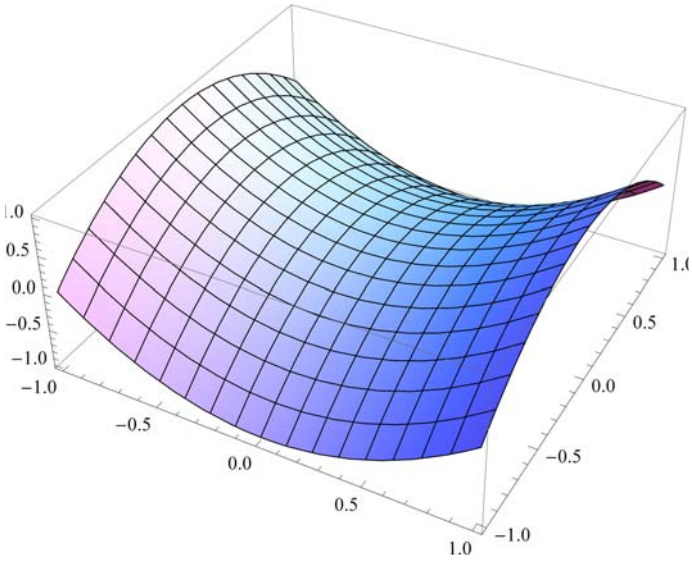
- draw graph of function f[x]

```
Plot[Sin[x], {x, -4 π, 4 π}]
```



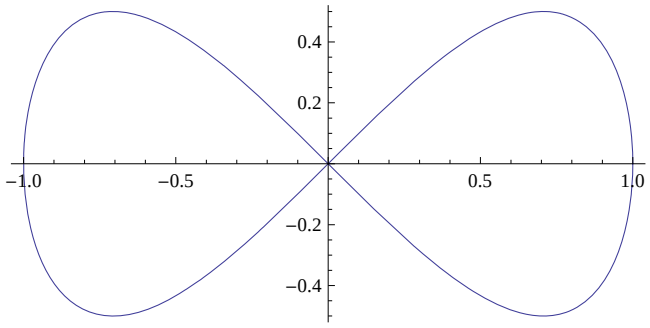
- draw graph of function f[x,y]

```
Plot3D[x^2 - y^2, {x, -1, 1}, {y, -1, 1}]
```



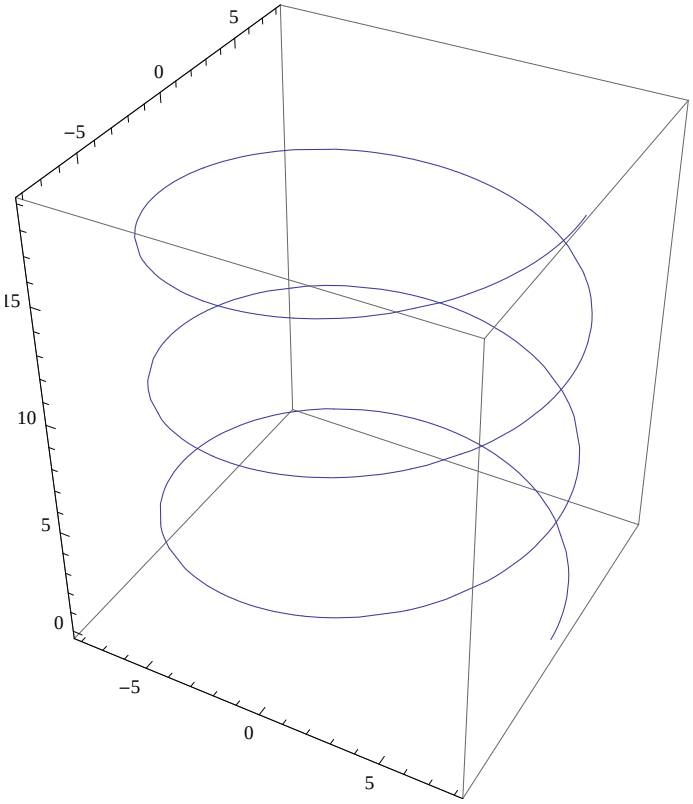
- draw curve {x[t], y[t]}

```
ParametricPlot[{Cos[t], Sin[t] Cos[t]}, {t, 0, 2 π}]
```



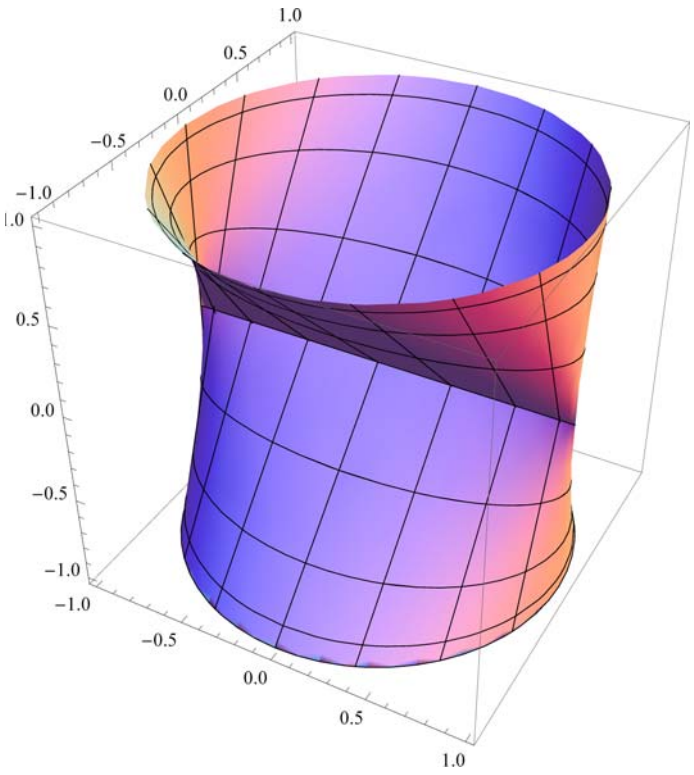
- draw curve {x[t], y[t], z[t]}

```
ParametricPlot3D[{8 Cos[t], 8 Sin[t], t}, {t, 0, 6 π}]
```



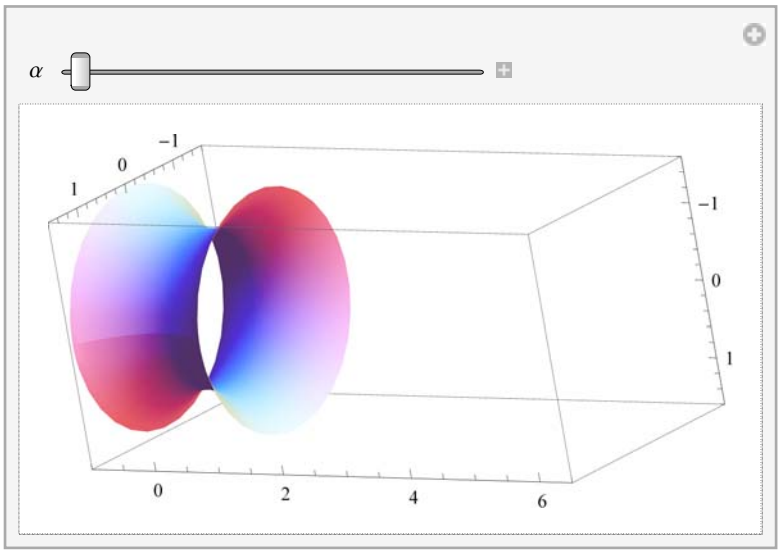
- draw surface {x[u,v], y[u,v], z[u,v]}

```
ParametricPlot3D[{Cos[u], Sin[u] Cos[v], Cos[v]}, {u, 0, 2 π}, {v, 0, 2 π}]
```



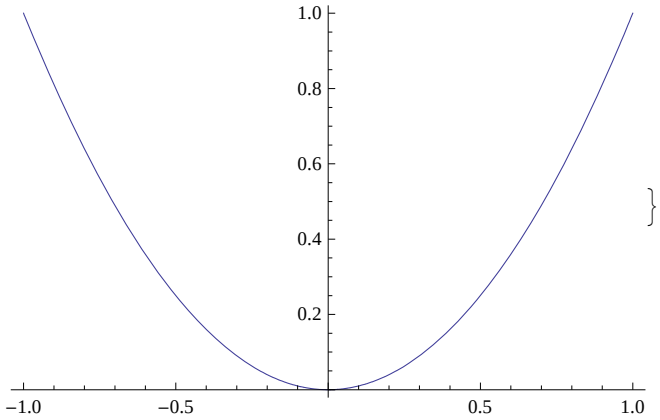
- visualize parametric dependance

```
Manipulate[ParametricPlot3D[Cos[α] {Cosh[u] Cos[v], Cosh[u] Sin[v], u} +
    Sin[α] {Sinh[u] Sin[v], -Sinh[u] Cos[v], v}, {u, -1, 1}, {v, 0, 2 π},
  PlotRange → {{-1.6, 1.6}, {-1.6, 1.6}, {-1, 6.5}}, ViewPoint → {66, -22, 54},
  ViewVertical → {-1.8, -0.8, -0.5}, Mesh → None,
  PlotPoints → 30],
 {α, 0, π/2}]
```



■ One of the most important basic objects is List[a, b, ...] = {a, b, ...}

```
ls = {a, Sin[x], 2^z, Plot[x^2, {x, -1, 1}]}
```



- access to a component

```
ls[[2]]
```

```
Sin[x]
```

```
{a, b, {{c, e}, d}, f}[[3, 1, 2]]
```

```
e
```

- algebraic operations
  - addition

```
2 {a, b, c} + 3 {x, y, z}
```

```
{2 a + 3 x, 2 b + 3 y, 2 c + 3 z}
```

  - multiplication

```
{a, b, c} {x, y, z}
```

```
{a x, b y, c z}
```

  - inner product

```
{a, b, c}.{x, y, z}
```

```
{{a, b}, {c, d}}.{x, y}
```

```
{a x + b y, c x + d y}
```

- visualization of lists

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$

```
{{a x + b y}, {c x + d y}}
```

```
MatrixForm[{{a, b}, {c, d}}.{x, y}]
```

$$\begin{pmatrix} a\,x + b\,y \\ c\,x + d\,y \end{pmatrix}$$

Combinatorial computational geometry

- **The main reference**

F.P.Preparata and M.I.Shamos Computational Geometry. Springer - Verlag, New York (1985).

Convex Hull

A subset H of $\mathbb{R}^n$ is called *convex* if for any points A, B $\in$ H the segment [A, B] belongs to H. Given M $\subset \mathbb{R}^n$, the smallest convex set H $\subset \mathbb{R}^n$ containig M is called the *convex hull* of M.

Exercise. Prove that the convex hull of M conicides with the intersection of all half-spaces containing M.

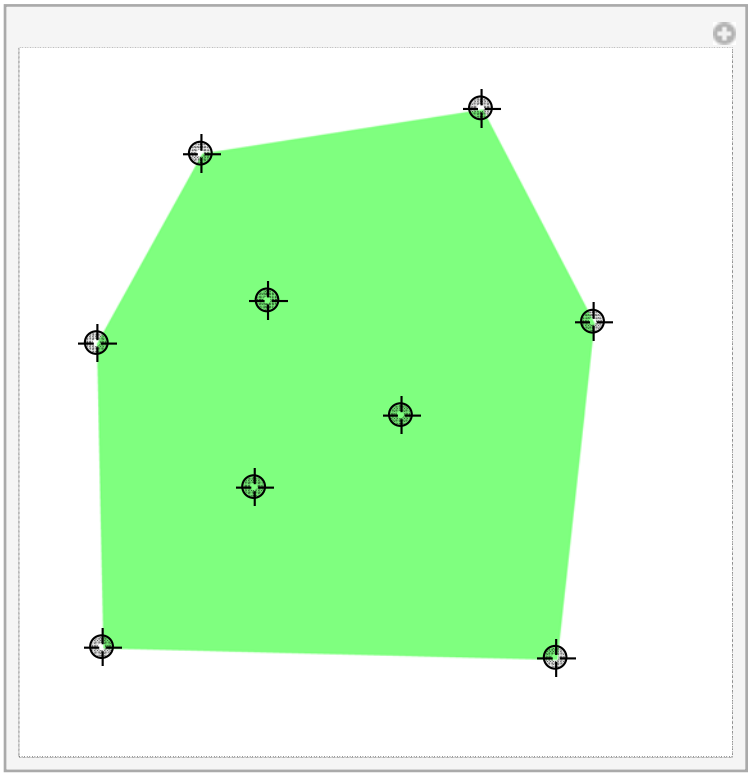
In two dimensions, the convex hull is found conceptually by stretching a rubber band around the points so that all of the points lie within the band. In multidimensions, it is not true.

Algorithms (Brute force, Graham Scan, Jarvis's march (gift wrapping), Divide-and-Conquer, Quick hull, Chan's algorithm(output sensitivity), Incremental, Andrew's Monotone Chain Algorithm)

http : // softsurfer.com/Archive/algorithm_ 0109/algorithm_ 0109. htm
http://fleischer.selfip.com/Courses/Algorithms/Alg_cs_07w/Webprojects/Zhaobo_hull/index.html#section31

Mathematica Implementation

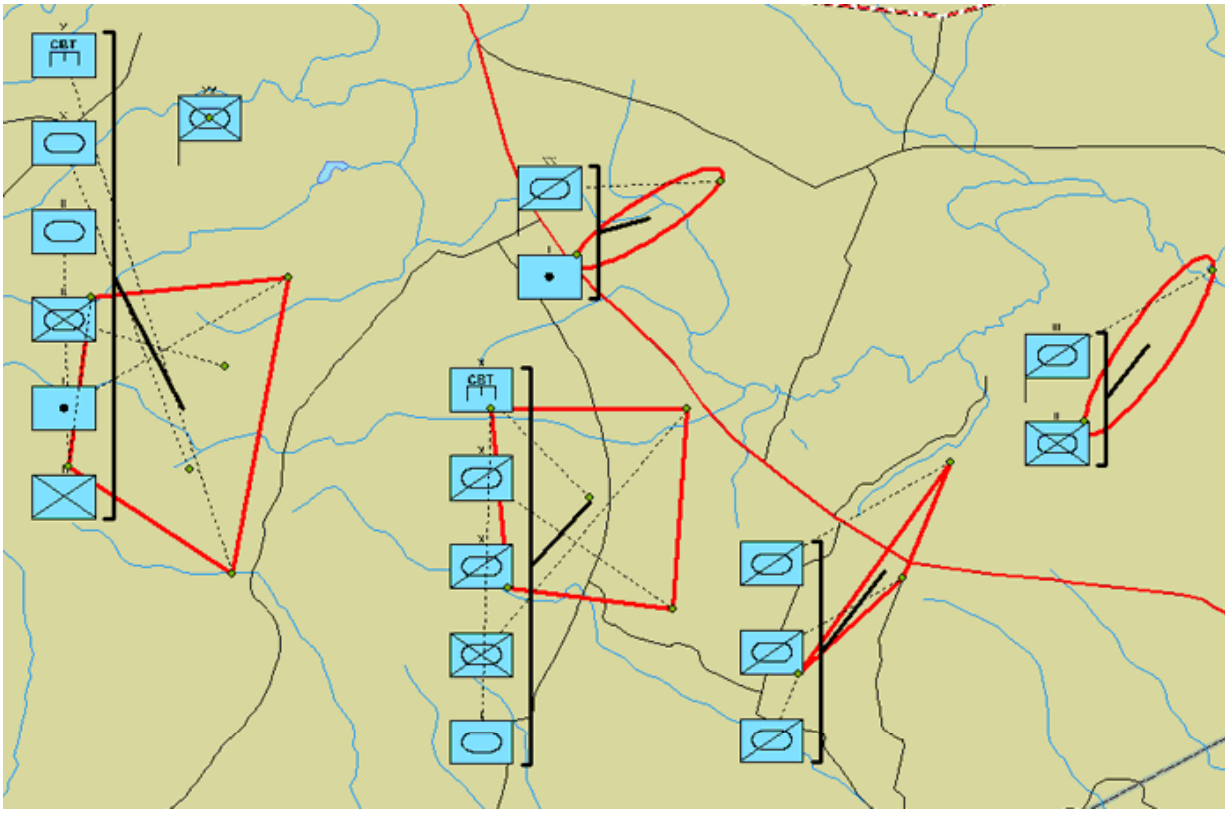
```
Needs["ComputationalGeometry`"];

Manipulate[Graphics[{Green, Opacity[0.5], Polygon[pp[[ConvexHull[pp]]]]},
  PlotRange → {{-0.5, 1.5}, {-0.5, 1.5}}],
 {{pp, {{0, 0}, {1, 0}, {0, 1}}}, Locator, LocatorAutoCreate → True}, SaveDefinitions → True]
```



Applications

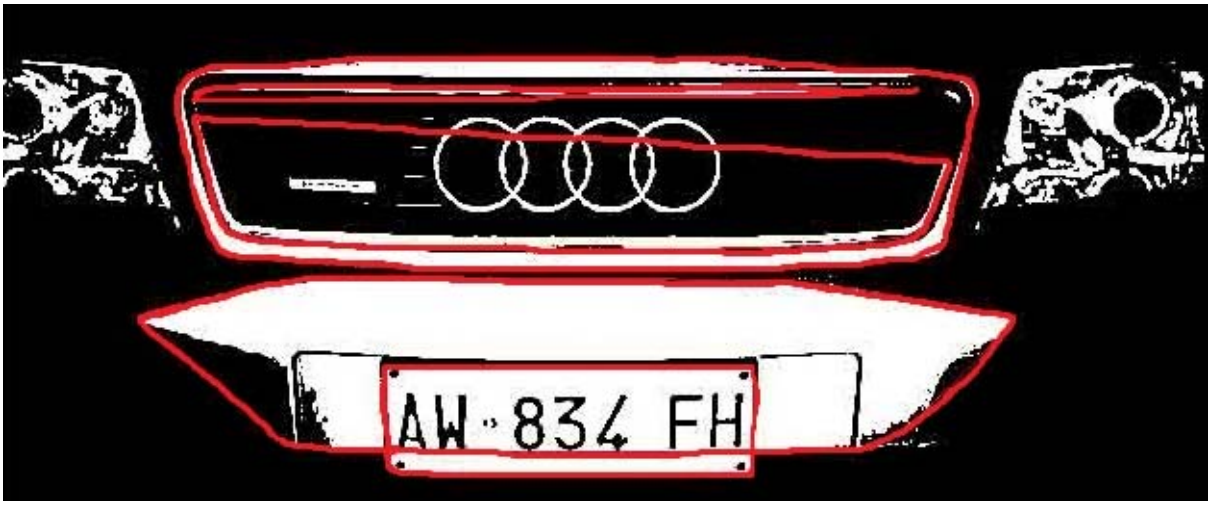
(1) Computer visualization, ray tracing (e.g. video games, replacemant of bounding boxes).
Suppose it is required to find intersections in the set of geometrical objects. The standard initial check is to find the intersections between their bounding boxes (it allows to quickly exclude from checks the pairs that are far apart). Convex hull can be a candidate for replacement of bounding boxes in these situations.

(2) Geographical Information Systems (GIS) (e.g. computing accessibility maps).
Suppose we are constructing a military map. We know the placement of some military object, and we need to draw a zone with restricted access. To do that, the convex hull of the military objects is usually used.



(3) Visual Pattern Matching (e.g. detecting car license plates for monitoring traffic speeds, criminal activities, etc.)

Recognition process starts from determining convex hulls of vairous elements of the picture.



(4) Geometry (e.g. diameter computation)

Diameter of $M \subset \mathbb{R}^n$ is the smallest d such that for any x, y ∈ M we have distance (x, y) ≤ d. To find diameter of a finite M, one can start from finding the boundary ∂conv(M) of the convex hull conv(M) of M (the answer is the vertices set of the polygon ∂conv(M)), and than seek the diameter of this smaller set.

■ Description of *Mathematica* Implementation

```
(*
Mathematica packages (here it is "ComputationalGeometry`")
 are files written in theMathematica language.
  They consist of collections of Mathematica definitions which
 "teach" Mathematica about particular application areas.
  Needs["PackageName`"] loads an appropriate file if it is not already loaded
*)
Needs["ComputationalGeometry`"];

Manipulate[(* generates a version of expression with controls added
  to allow interactive manipulation of the value of the expression *)
 (* description of the object to manipulate with *)
 Graphics[(* show the graphical object described here *)
  {
   Green, (* the directive makes the object to be green *)
   Opacity[0.5], (* the directive makes the object to be
    transparent: 1 means nontransparent (opacity), 0 means completely transparent *)
   Polygon[ (* it is polygon whose consecutive vertices are
     given in the list {{x1,y2}, {x2,y2}, ... } *)
    pp[[(* if here {i1, i2, ... } stands, then this means {pp[[i1]], pp[[i2]], ... } *)
     ConvexHull[
      pp (* the list  {{x1,y1}, {x2,y2}, ... } of points {xi,yi} in the plane *)
     ] (* the list {i1, i2, ... } of numbers of consecutive
      points from pp forming the boundary of the convex hull of pp *)
    ]]
   ]
  },
  PlotRange → {{-0.5, 1.5}, {-0.5, 1.5}} (* is an option for graphics
   functions that specifies what range of coordinates to include in a plot;
  here Graphics draws only in rectangle -0.5 ≤ x ≤ 1.5 and -0.5 ≤ y ≤ 1.5 *)
 ],
 (* parameters of the object described above to change *)
 {{pp, (* the list of locators *)
   {{0, 0}, {1, 0}, {0, 1}} (* initial positions of
    locators: here we start from 3 locators at points {0,0}, {1,0}, and {0,1} *)
  },
  Locator, (* tells to manipulate that the parameter is a list of locators *)
  LocatorAutoCreate → True
  (* regulates possibility to create new or delete old locators by means of Alt+
   LeftClick; True means that it is allowed *)
 },
 SaveDefinitions → True (* tells to manipulate to save all relevant definitions to
  reconstruct the last state of the object in the next load of Mathematica *)
]
```

■ Playground with *Mathematica* functions

Task 1. Draw graphical primitives.

```
Graphics[Point[{1, 0.1}]]
```



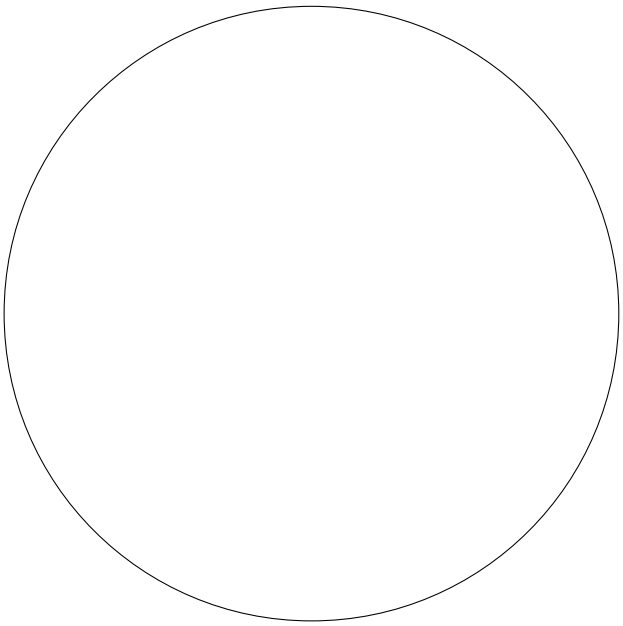
```
Graphics[Line[{{0, 0}, {1, 0.1}}]]
```

```
Graphics[Arrow[{{0, 0}, {1, 0.1}}]]
```

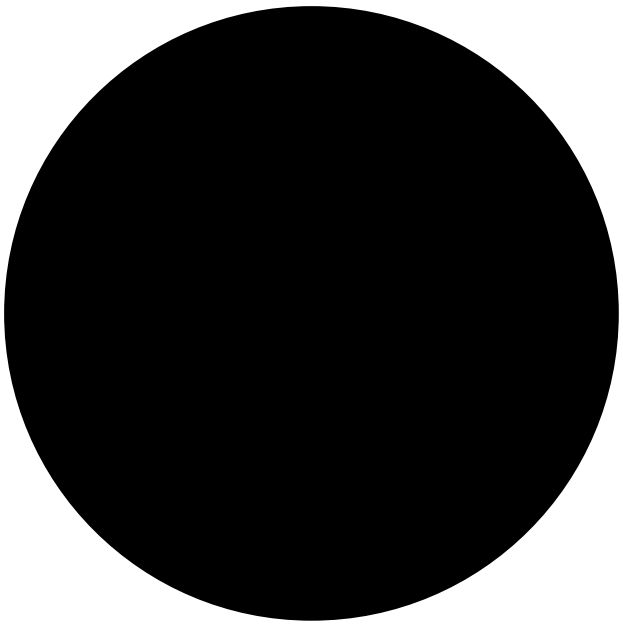
```
Graphics[Line[{{0, 0}, {1, 0.5}, {3, -0.1}, {4, 0.2}}]]
```

```
Graphics[Arrow[{{0, 0}, {1, 0.5}, {3, -0.1}, {4, 0.2}}]]
```

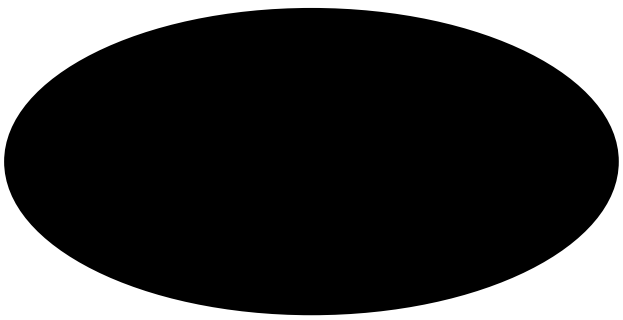
```
Graphics[Circle[{1, 2}, 1]]
```



```
Graphics[Disk[{1, 2}, 1]]
```

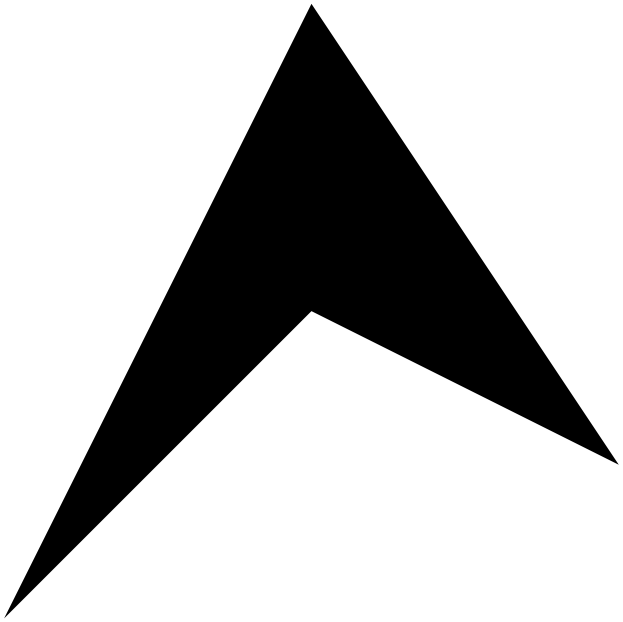


```
Graphics[Disk[{0, 0}, {2, 1}]]
```



```
Graphics[Rectangle[{0, 0}, {2, 1}]]
```

```
Graphics[Polygon[{{0, 0}, {1, 1}, {2, 0.5}, {1, 2}}]]
```



```
Graphics[Text["This is a text", {0, 1}]]
```

This is a text

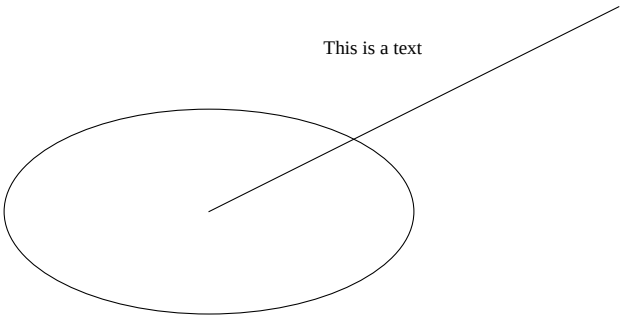
```
Graphics[Locator[{0.5, 0.5}]]
```

```
Graphics[BezierCurve[{{0, 0}, {1, 0.5}, {2, -0.5}, {3, 0.5}}]]
```

```
Graphics[BSplineCurve[{{0, 0}, {1, 0.5}, {2, -0.5}, {3, 0.5}}]]
```

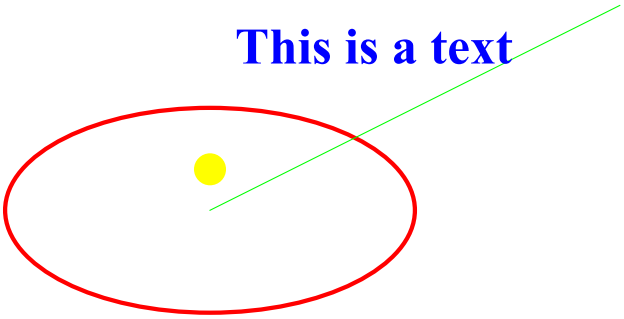
Task 2. Combine graphical primitives.

```
Graphics[{Circle[{-1, -0.5}, {1, 0.5}],
  Line[{{-1, -0.5}, {1, 0.5}}], Text["This is a text", {-0.2, 0.3}]}]
```



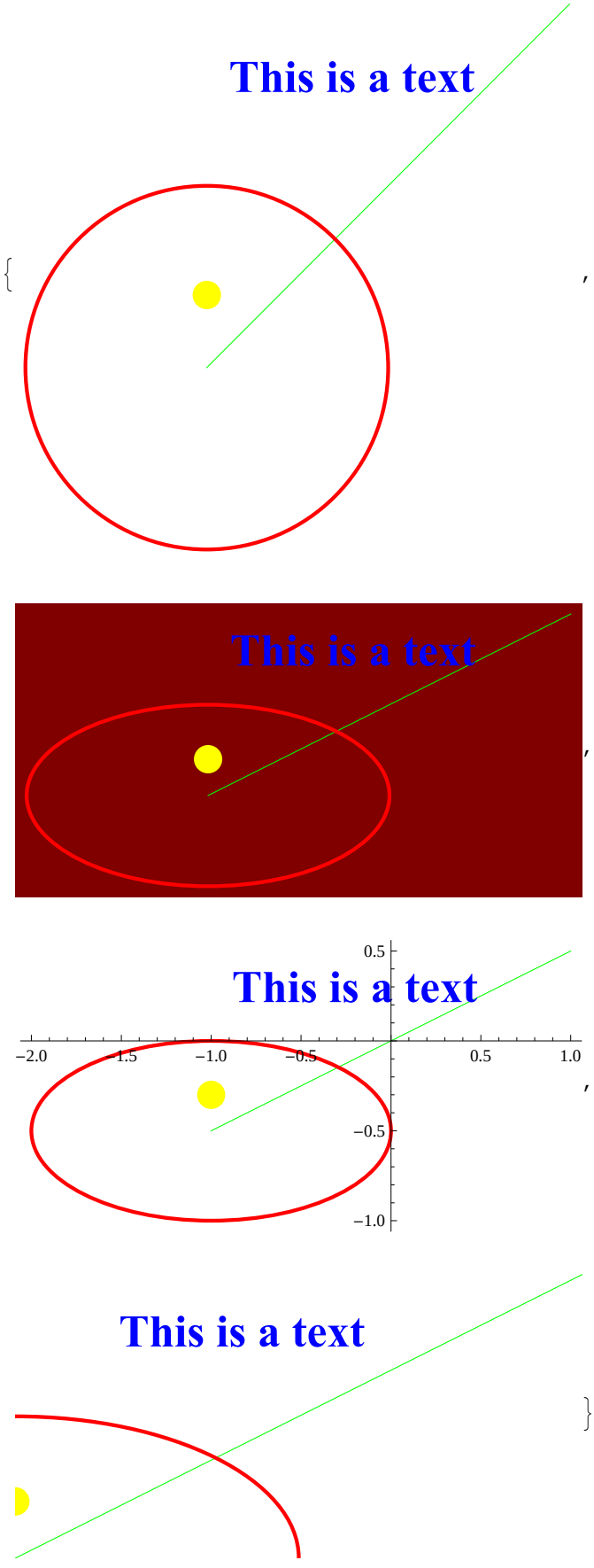
Task 3. Modify graphical primitives.

```
Graphics[{
  Yellow, PointSize[0.05], Point[{-1, -0.3}],
  Green, Line[{{-1, -0.5}, {1, 0.5}}],
  {Red, Thick, Circle[{-1, -0.5}, {1, 0.5}]},
  Text[Style["This is a text", Large, Bold, Blue], {-0.2, 0.3}]
 }]
```



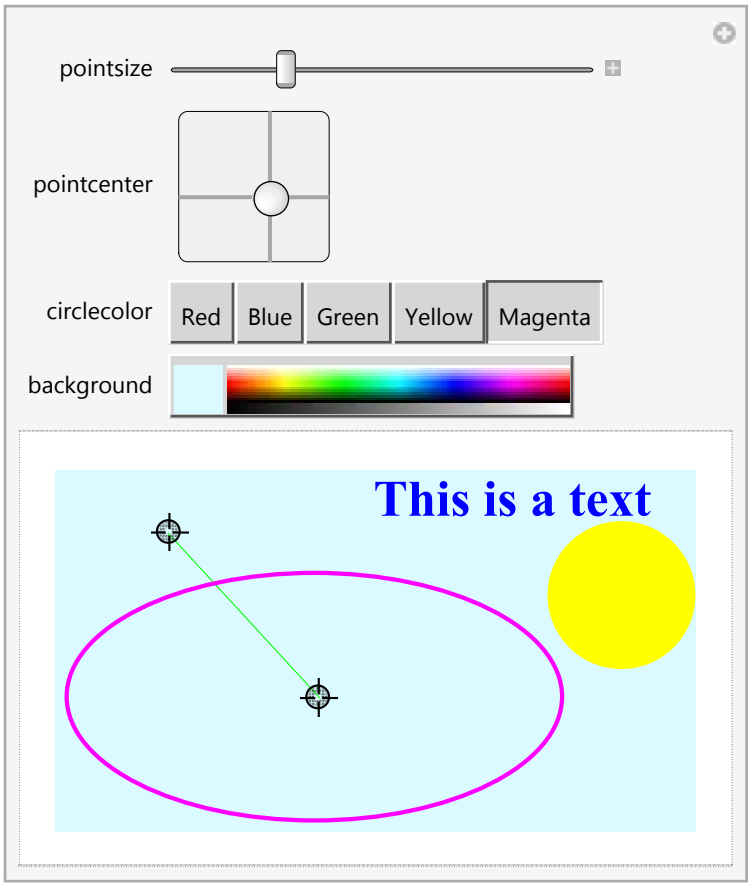
Task 4. Modify the entier picture.

```
{
 Graphics[{Yellow, PointSize[0.05], Point[{-1, -0.3}], Green,
   Line[{{-1, -0.5}, {1, 0.5}}], {Red, Thick, Circle[{-1, -0.5}, {1, 0.5}]},
   Text[Style["This is a text", Large, Bold, Blue], {-0.2, 0.3}]},
  AspectRatio → 1 (* the ratio of height to width for a plot *)],
 Graphics[{Yellow, PointSize[0.05], Point[{-1, -0.3}], Green,
   Line[{{-1, -0.5}, {1, 0.5}}], {Red, Thick, Circle[{-1, -0.5}, {1, 0.5}]},
   Text[Style["This is a text", Large, Bold, Blue], {-0.2, 0.3}]},
  Background → RGBColor[0.5, 0, 0] (* background color to use *)],
 Graphics[{Yellow, PointSize[0.05], Point[{-1, -0.3}], Green,
   Line[{{-1, -0.5}, {1, 0.5}}], {Red, Thick, Circle[{-1, -0.5}, {1, 0.5}]},
   Text[Style["This is a text", Large, Bold, Blue], {-0.2, 0.3}]},
  Axes → True (* whether axes should be drawn *)],
 Graphics[{Yellow, PointSize[0.05], Point[{-1, -0.3}], Green,
   Line[{{-1, -0.5}, {1, 0.5}}], {Red, Thick, Circle[{-1, -0.5}, {1, 0.5}]},
   Text[Style["This is a text", Large, Bold, Blue], {-0.2, 0.3}]},
  PlotRange → {{-1, 1}, {-0.5, 0.5}} (* what range of
   coordinates to include in a plot *)]
}
```

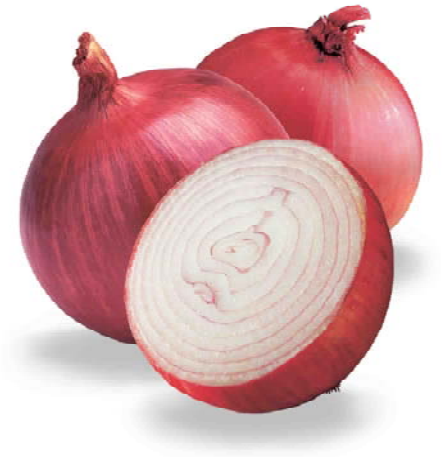


Task 5. Manipulate picture.

```
Manipulate[
 Graphics[{Yellow, PointSize[pointsize], Point[pointcenter],
   Green, Line[p], {circlecolor, Thick, Circle[{-1, -0.5}, {1, 0.5}]}, Text[
    Style["This is a text", Large, Bold, Blue], {-0.2, 0.3}]}, Background → background],
 {{pointsize, 0.05}, 0.01, 1}, (* slider *)
 {{pointcenter, {-1, -0.3}}, {-1, -0.5}, {1, 0.5}}, (* 2D-slider *)
 {{circlecolor, Red},
  {Red → "Red", Blue → "Blue", Green → "Green", Yellow → "Yellow", Magenta → "Magenta"}},
 (*  setter bar for few elements;popup menu for more; here it's for colors *)
 {{p, {-1, -0.5}, {1, 0.5}}}, Locator}, (* Locator *)
 {background, White} (* color slider *)
]
```



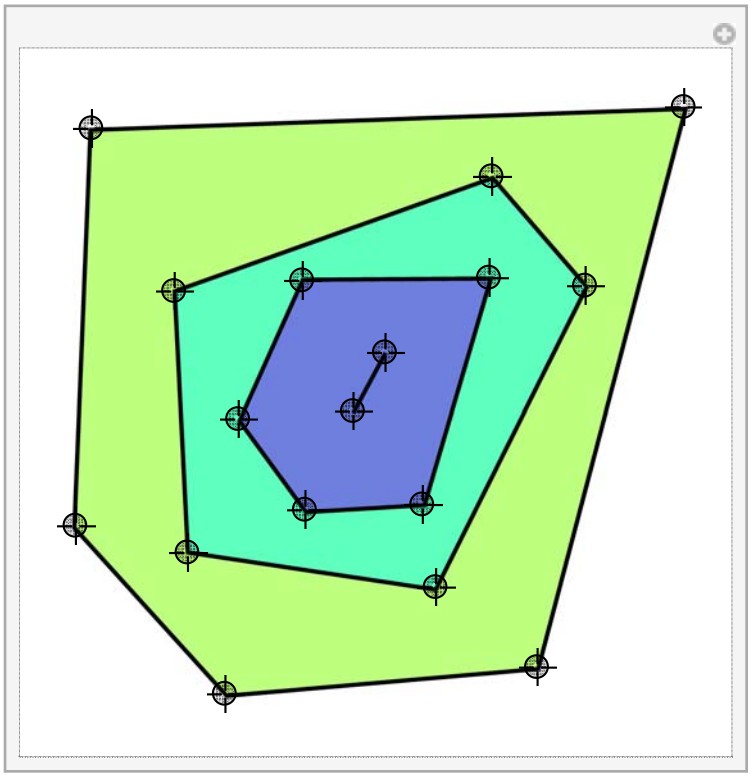
■ Generalization: Onion Peelings (Convexity Levels)



Given $M \subset \mathbb{R}^n$, the *first convexity level* (*first onion peeling*) $\text{conv}_1(M)$ of M is the set of all points from M lying on the boundary of the conv(M). The *k-th convexity level* $\text{conv}_k(M)$ of M is the first convexity level of $M \setminus \bigcup_{i<k} \text{conv}_i(M)$. The number of convexity levels of the set M is called the *height* of M.

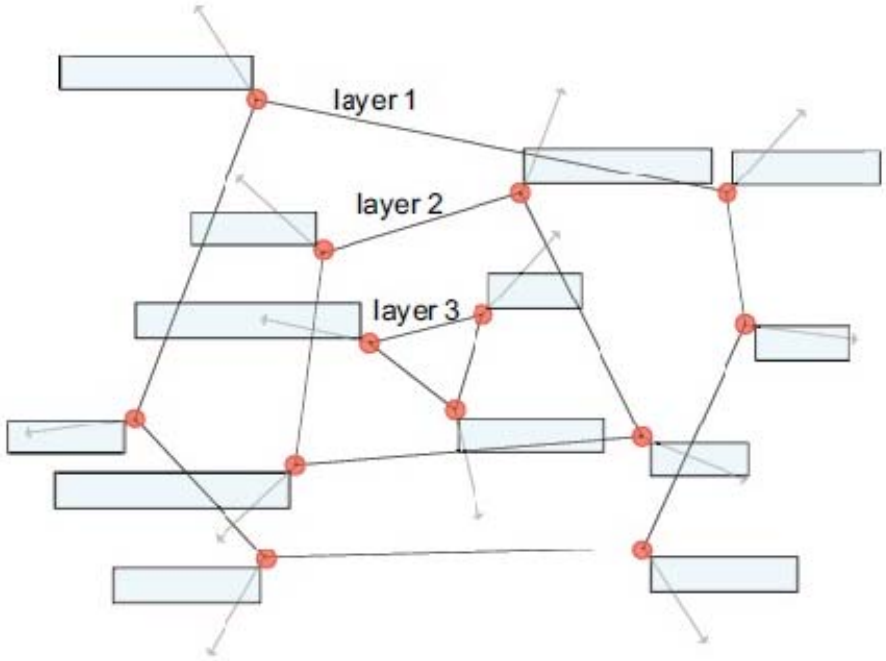
■ *Mathematica* Implementation

```
Manipulate[
 Module[{res, pts, ch, n},
  pts = pp;
  res = {};
  While[pts ≠ {}, ch = ConvexHull[pts];
   res = res ~Join~ {pts[[ch]]}; pts = Delete[pts, Partition[ch, 1]]];
  n = Length[res];
  Graphics[
   Flatten@Table[{Hue[i/n], Opacity[0.5], EdgeForm[Thick], Polygon[res[[i]]]}, {i, n}],
   PlotRange → {{-0.5, 1.5}, {-0.5, 1.5}}]
 ],
 {{pp, {{0, 0}, {1, 0}, {0, 1}}}, Locator, LocatorAutoCreate → True}]
```



■ **Applications**

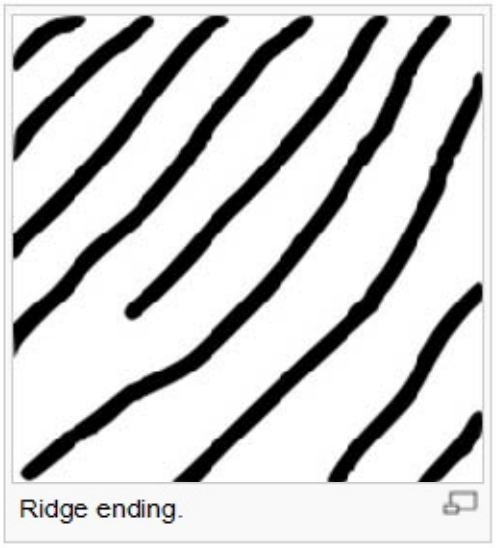
(1) Map Labeling. Suppose we need to label different cites in a map in such a way that the label do not intersect each other, do not cover other cites and look aesthetic nice in whole.



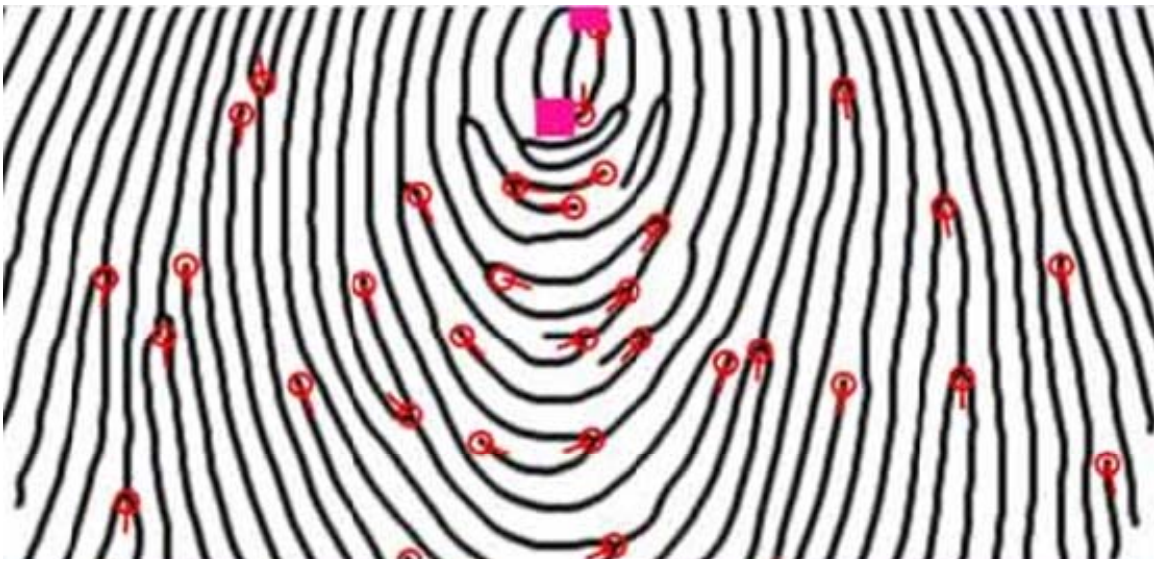
(2) Fingerprint recognition

- Fingerprint types

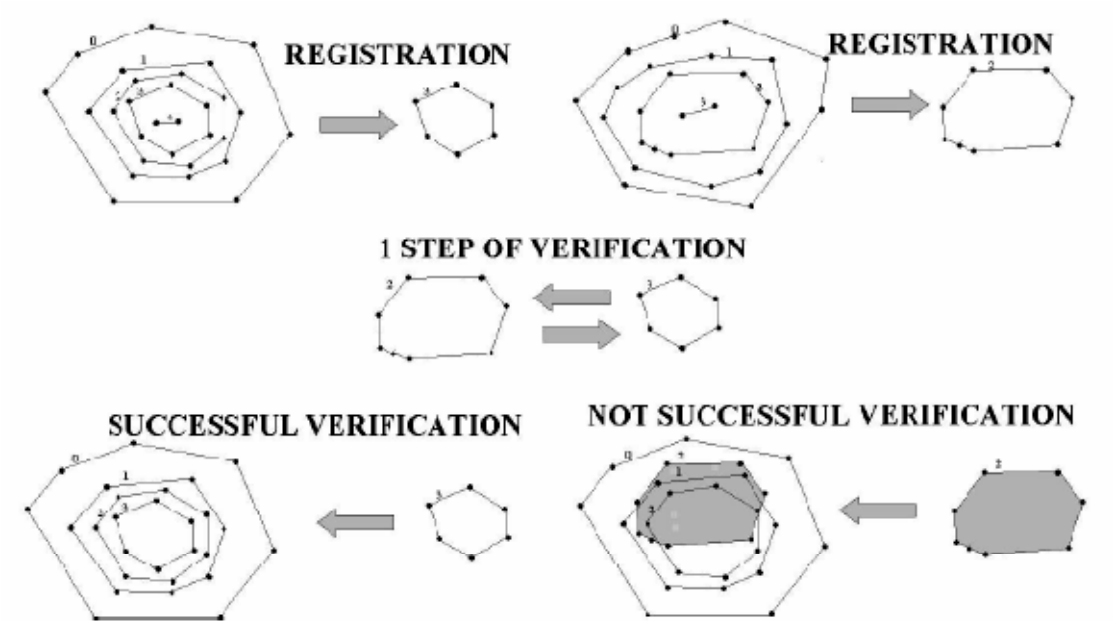
- Minutiae



- A typical live - scan fingerprint will contain 30 - 40 minutiae

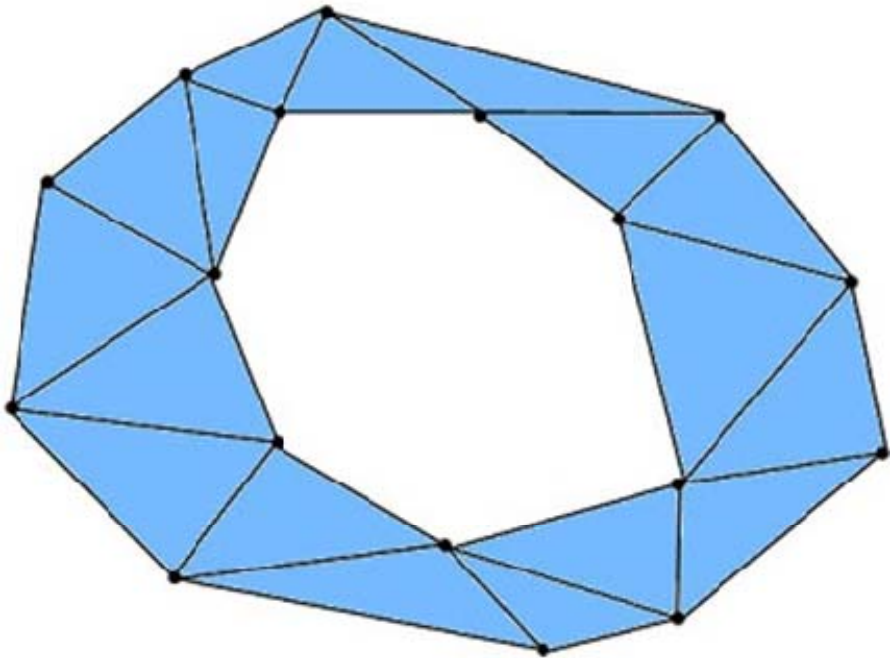


- Compare onion peelings



- **Exercise**

1) Construct Onion triangulation



2) Create a program that determines if a point lies in the convex hull of a given point set or not.

■ Description of *Mathematica* Implementation

```
Manipulate[
 Module[{res, pts, ch, n}, (* specifies that occurrences of the symbols res,
  pts, ch, n should be treated as local *)
  pts = pp; (* assingning to pst the list pp of locators *)
  res = {}; (* assigning to res the emply list {} *)
  While[pts ≠ {}, (* it works until the condition pts≠{} holds *)
   ch = ConvexHull[pts]; (* ch is the list of indices of consecutive
    points from pts lying on the boundary of the convex hull of pts *)
   res = res ~Join~ {pts〚ch〛}; (* we add the list of points
    with indices ch to the list res *)
   pts = Delete[pts, Partition[ch, 1]] (* we delete the
    previous points pts〚ch〛 from pts *)
  ];
  n = Length[res]; (* assingn to n the number of elements in the list res *)
  Graphics[(* draw polygons (convesity levels, onion peelings) in different colors *)
   Flatten@Table[(* create the list of n elements-polygons to be drawn *)
     {Hue[i/n], (* directive Hue[x], 0 ≤ x ≤ 1, specifies the corresponding color *)
      Opacity[0.5],
      EdgeForm[Thick], (* this directive regulate the
       form of the boundary of polygon; the boundary will be thick *)
      Polygon[res〚i〛] (* polygon spanned by consecutive
       points from the list res〚i〛 *)
     },
     {i, n}
    ], PlotRange → {{-0.5, 1.5}, {-0.5, 1.5}}
   ]
  ],
 {{pp, {{0, 0}, {1, 0}, {0, 1}}}, Locator, LocatorAutoCreate → True}]
```

■ Playground with *Mathematica* functions

Task 1. Show that variables x, y ... in Module[{x, y, ...}, ...] are local, but all other not.

```
Module[{x}, x = 1; y = 2];
{x, y}
```

```
{x, 2}
```

```
y
Clear[y] (* makes y free *)
y
```

```
2
```

```
y
```

Task 2. Using While, calculate 50!.

```
Module[{n = 1, res = 1}, While[n ≤ 50, res *= n; n++]; res]
(* res*=n means res=res*n;  n++ means n=n+1 *)
50!
```

```
30 414 093 201 713 378 043 612 608 166 064 768 844 377 641 568 960 512 000 000 000 000
```

```
30 414 093 201 713 378 043 612 608 166 064 768 844 377 641 568 960 512 000 000 000 000
```

Task 3. Partition the list {1, 2, ... 100} for consecutive sublists of the length 5.

```
Range[100]
Partition[Range[100], 5]
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41,
 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61,
 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81,
 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100}
```

```
{{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}, {16, 17, 18, 19, 20},
 {21, 22, 23, 24, 25}, {26, 27, 28, 29, 30}, {31, 32, 33, 34, 35}, {36, 37, 38, 39, 40},
 {41, 42, 43, 44, 45}, {46, 47, 48, 49, 50}, {51, 52, 53, 54, 55}, {56, 57, 58, 59, 60},
 {61, 62, 63, 64, 65}, {66, 67, 68, 69, 70}, {71, 72, 73, 74, 75}, {76, 77, 78, 79, 80},
 {81, 82, 83, 84, 85}, {86, 87, 88, 89, 90}, {91, 92, 93, 94, 95}, {96, 97, 98, 99, 100}}
```

Task 4. Delete from $\{a_1, a_2, \ldots, a_{10}\}$ all elements with prime indices.

```
Delete[{a₁, a₂, a₃, a₄, a₅, a₆, a₇, a₈, a₉, a₁₀}, {{2}, {3}, {5}, {7}}]
```

$\{a_1, a_4, a_6, a_8, a_9, a_{10}\}$

Task 5. Delete from $\{a_1, a_2, \ldots, a_{100}\}$ all elements with prime indices.

```
Module[{prime = {}, i = 1, next = Prime[i], ind},
 For[i = 1, (next = Prime[i]) ≤ 100, i++, prime = prime ~Join~ {next}];
 ind = Partition[prime, 1];
 Print[ind];
 Delete[Table[aᵢ, {i, 100}], ind]
]
```

```
{{2}, {3}, {5}, {7}, {11}, {13}, {17}, {19}, {23}, {29}, {31}, {37},
 {41}, {43}, {47}, {53}, {59}, {61}, {67}, {71}, {73}, {79}, {83}, {89}, {97}}
```

$\{a_1, a_4, a_6, a_8, a_9, a_{10}, a_{12}, a_{14}, a_{15}, a_{16}, a_{18}, a_{20}, a_{21}, a_{22}, a_{24}, a_{25}, a_{26}, a_{27}, a_{28}, a_{30},$
$a_{32}, a_{33}, a_{34}, a_{35}, a_{36}, a_{38}, a_{39}, a_{40}, a_{42}, a_{44}, a_{45}, a_{46}, a_{48}, a_{49}, a_{50}, a_{51}, a_{52}, a_{54}, a_{55},$
$a_{56}, a_{57}, a_{58}, a_{60}, a_{62}, a_{63}, a_{64}, a_{65}, a_{66}, a_{68}, a_{69}, a_{70}, a_{72}, a_{74}, a_{75}, a_{76}, a_{77}, a_{78},$
$a_{80}, a_{81}, a_{82}, a_{84}, a_{85}, a_{86}, a_{87}, a_{88}, a_{90}, a_{91}, a_{92}, a_{93}, a_{94}, a_{95}, a_{96}, a_{98}, a_{99}, a_{100}\}$

Task 6. Convert a matrix 3 x 3 into vector.

```
Flatten[{{a, b, c}, {d, e, f}, {g, h, i}}]
```

$$\text{Flatten}\left[\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}\right]$$

$\{a, b, c, d, e, f, g, h, i\}$

$\{a, b, c, d, e, f, g, h, i\}$

Task 7. Generate the list of cubes of numbers from 5 to 100 in steps of 5 .

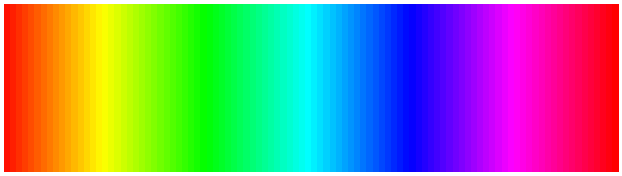
```
Table[i³, {i, 5, 100, 5}]
```

```
{125, 1000, 3375, 8000, 15 625, 27 000, 42 875, 64 000, 91 125, 125 000, 166 375,
 216 000, 274 625, 343 000, 421 875, 512 000, 614 125, 729 000, 857 375, 1 000 000}
```

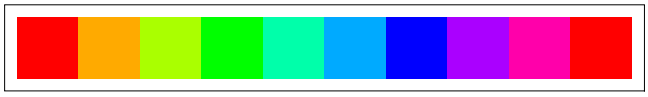
Task 8. Investigate the colors given by Hue[x] .

```
Manipulate[Graphics[{Hue[x], Rectangle[]}, AspectRatio → 0.3], {x, 0, 1}]
```


```
Graphics[Raster[{Table[i/100, {i, 100}]}, ColorFunction → Hue], AspectRatio → 0.3]
```



```
ArrayPlot[{Table[i/9, {i, 0, 9}]}, ColorFunction → Hue]
```



```
ArrayPlot[{Table[i/9, {i, 0, 9}]}, ColorFunction → "Rainbow"]
```

Task 7. Visualize the number π drawing its digits colors along a spiral

```
3.14159265358979323846264338327950288
```

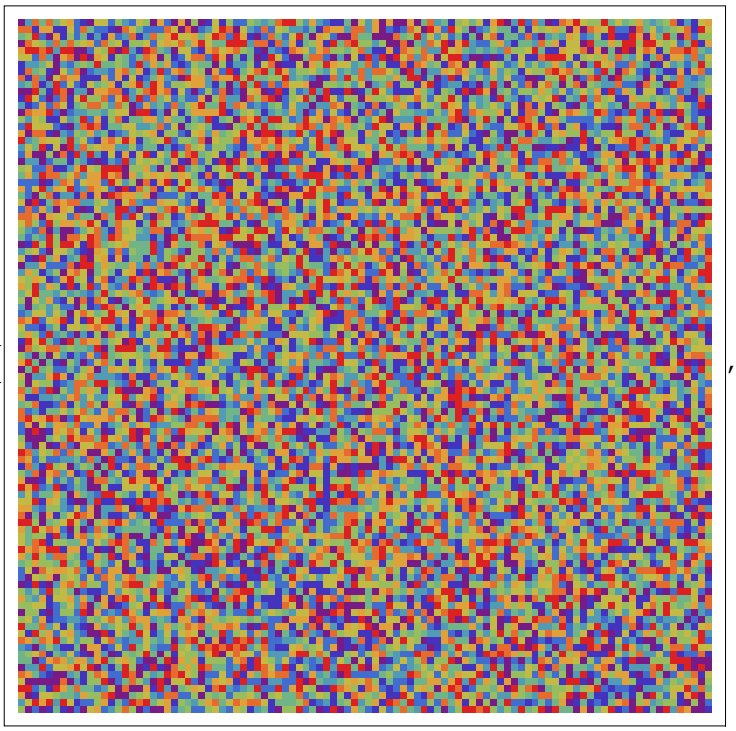
$$\begin{pmatrix} 6 & 4 & 8 & 3 & 2 & 8 \\ 2 & 2 & 9 & 5 & 3 & 8 \\ 6 & 6 & 3 & 1 & 9 & 2 \\ 4 & 5 & 1 & 4 & 7 & 0 \\ 3 & 3 & 5 & 8 & 9 & 5 \\ 3 & 8 & 3 & 2 & 7 & 9 \end{pmatrix}$$

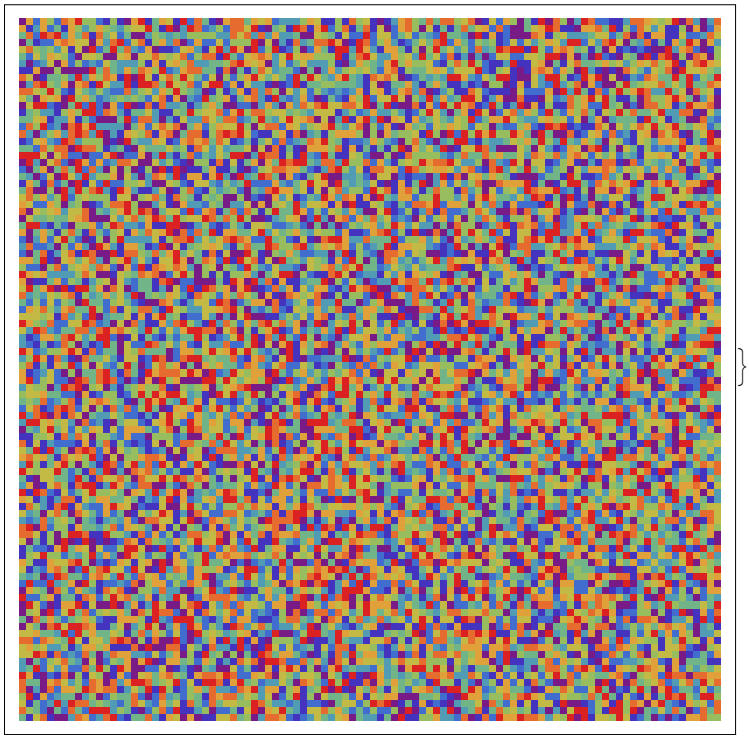
```
RealDigits[π, 10, 20]
(* the answer is { the first 20 digits of base 10 for the number π,
  the number of digits in the integer part } *)
```

```
{{3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3, 2, 3, 8, 4}, 1}
```

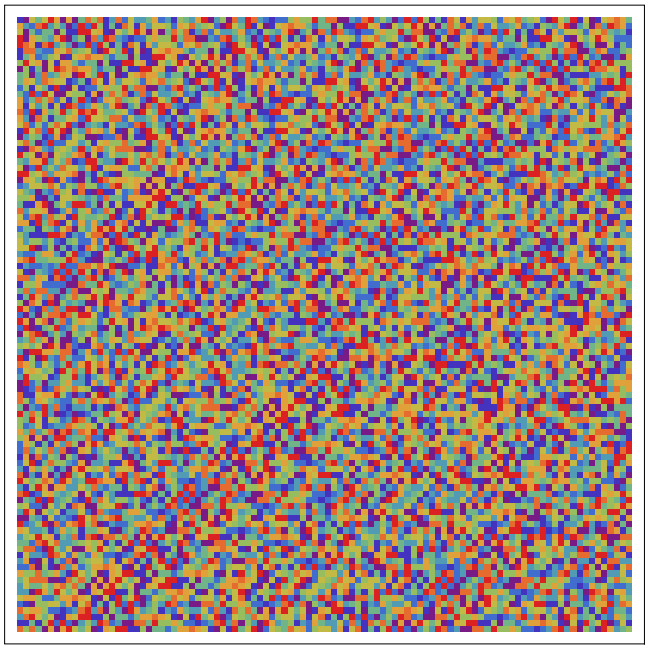
```
curpet[x_, n_, b_] := Module[
   {seq, m, mat,
    alldir = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}},
    numdir = 1, mult, cur, i = 2, j},
   m = 2 n;
   mat = Table[{i, j}, {i, m}, {j, m}];
   seq = RealDigits[N[x, $MaxPrecision], b, m^2][[1]];
   cur = {n, n};
   mat[[Sequence @@ cur]] = seq[[1]];
   mult = 1;
   While[i ≤ m^2,
    For[j = 1, j ≤ mult, j++,
     cur += alldir[[numdir]];
     mat[[Sequence @@ cur]] = seq[[i]];
     i++;
     If[i > m^2, Goto[end]];
    ];
    numdir = 1 + Mod[numdir, 4];
    For[j = 1, j ≤ mult, j++,
     cur += alldir[[numdir]];
     mat[[Sequence @@ cur]] = seq[[i]];
     i++;
     If[i > m^2, Goto[end]];
    ];
    numdir = 1 + Mod[numdir, 4];
    mult++;
   ];
   Label[end];
   ArrayPlot[mat, ColorFunction → "Rainbow", ColorFunctionScaling → True]
  ];
```

```
{curpet[π, 50, 10], curpet[e, 50, 10]}
```



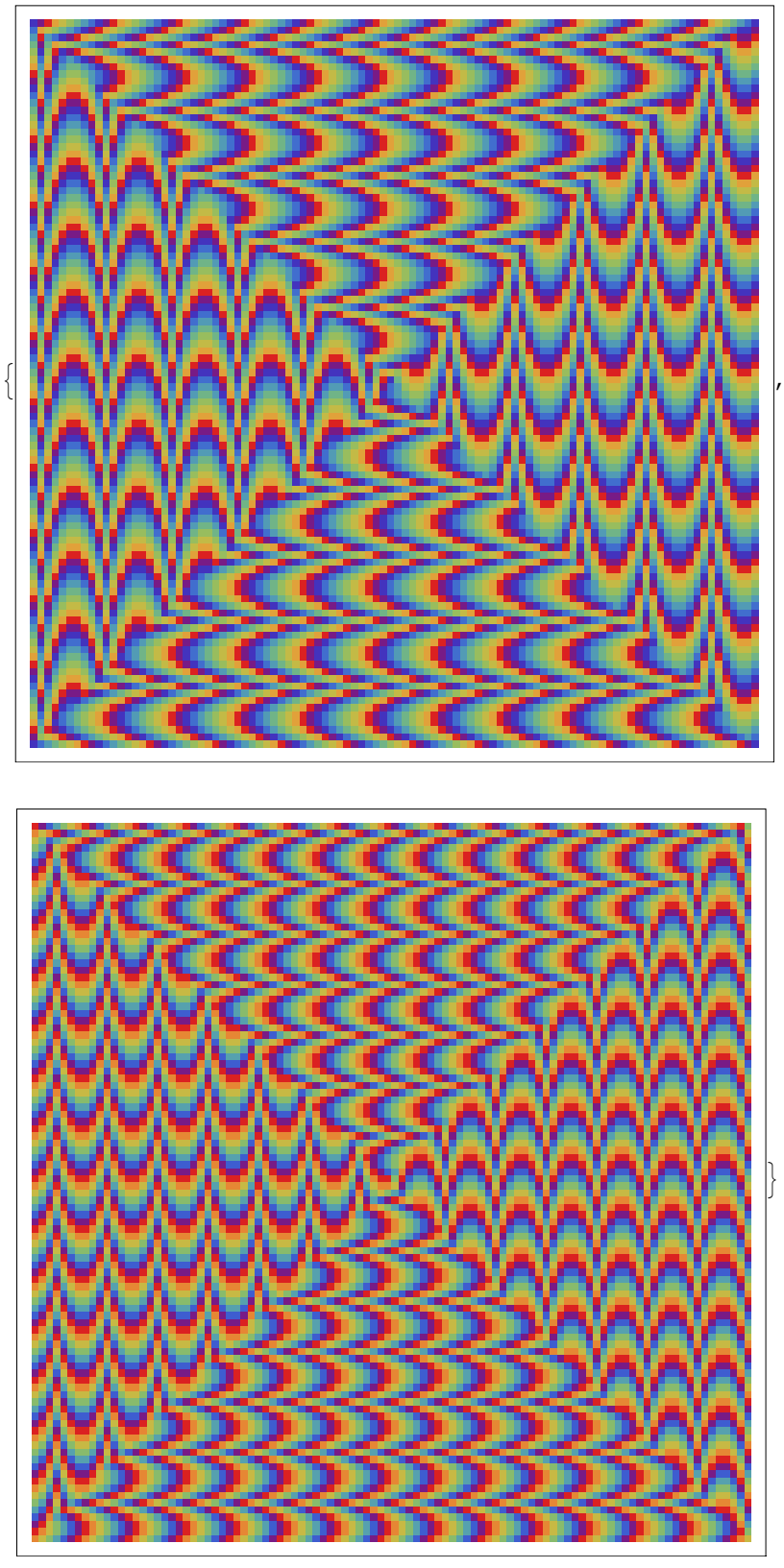


```
curpet[355 / 113, 50, 10]
```

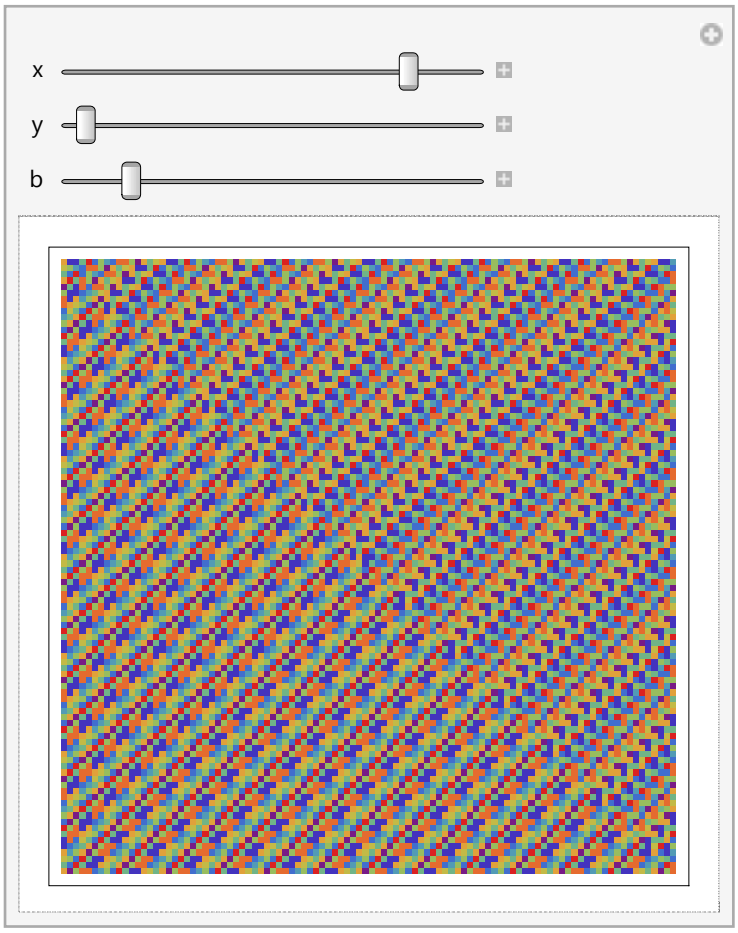


1/81 base 10 is similar to 1/49 base 43

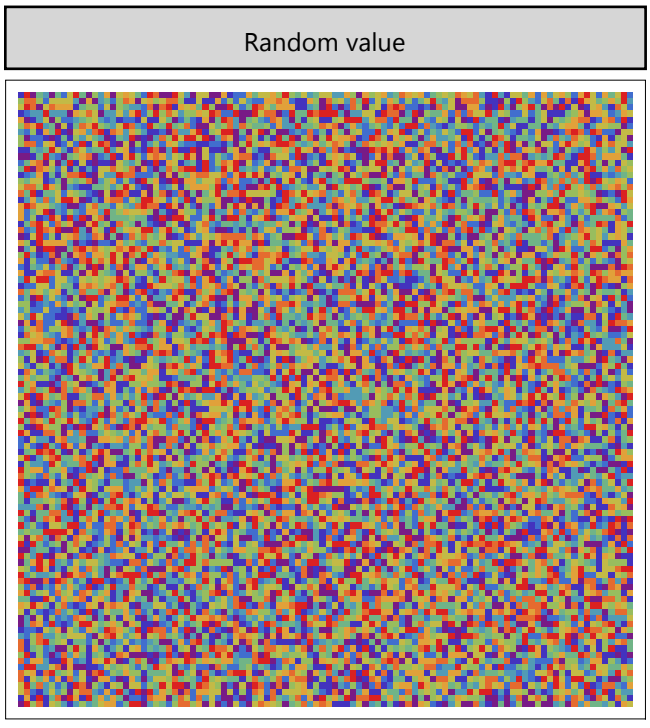
```
{curpet[1 / 81, 50, 10], curpet[1 / 49, 50, 43]}
```



```
Manipulate[curpet[y / x, 50, b], {{x, 81}, 1, 1000, 1},
 {y, 1, x, 1}, {{b, 10}, 2, 64, 1}, SaveDefinitions → True]
```




```
DynamicModule[{x = 0}, Grid[{{Button["Random value", x = Random[Real, {0, 1}, 10 000]]},
   {Dynamic[curpet[x, 50, 10]]}}]]
```

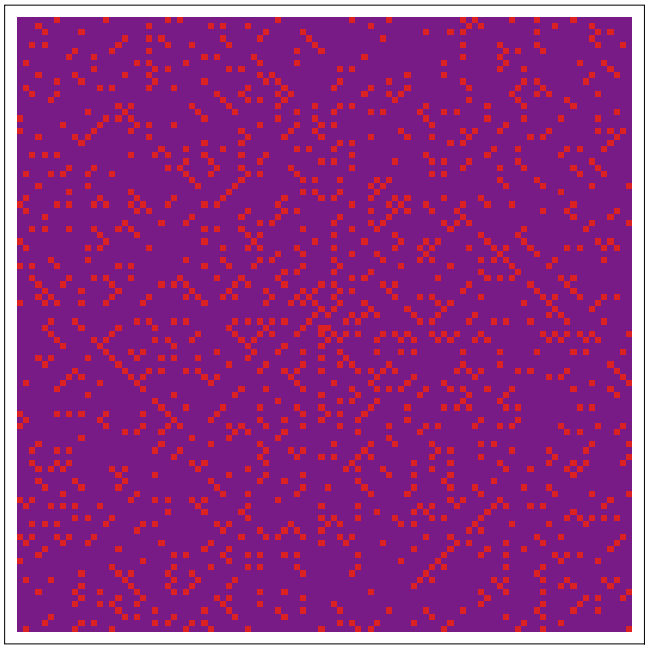


Task 7. Visualize distribution of prime numbers drawing the values of prime-indicator function of consecutive integers along a spiral

```
{{73, 72, 71, 70, 69, 68, 67, 66, 65, 100},
 {74, 43, 42, 41, 40, 39, 38, 37, 64, 99},
 {75, 44, 21, 20, 19, 18, 17, 36, 63, 98},
 {76, 45, 22, 7, 6, 5, 16, 35, 62, 97},
 {77, 46, 23, 8, 1, 4, 15, 34, 61, 96},
 {78, 47, 24, 9, 2, 3, 14, 33, 60, 95},
 {79, 48, 25, 10, 11, 12, 13, 32, 59, 94},
 {80, 49, 26, 27, 28, 29, 30, 31, 58, 93},
 {81, 50, 51, 52, 53, 54, 55, 56, 57, 92},
 {82, 83, 84, 85, 86, 87, 88, 89, 90, 91}}
```

```
curpetOfPrimes[n_] := Module[
   {myPrimeQ, seq, m, mat,
    alldir = {{1, 0}, {0, 1}, {-1, 0}, {0, -1}},
    numdir = 1, mult, cur, i = 2, j},
   myPrimeQ[x_] := If[PrimeQ[x], 1, 0];
   m = 2 n;
   mat = Table[{i, j}, {i, m}, {j, m}];
   seq = Range[m^2];
   cur = {n, n};
   mat[[Sequence @@ cur]] = myPrimeQ[seq[[1]]];
   mult = 1;
   While[i ≤ m^2,
    For[j = 1, j ≤ mult, j++,
     cur += alldir[[numdir]];
     mat[[Sequence @@ cur]] = myPrimeQ[seq[[i]]];
     i++;
     If[i > m^2, Goto[end]];
    ];
    numdir = 1 + Mod[numdir, 4];
    For[j = 1, j ≤ mult, j++,
     cur += alldir[[numdir]];
     mat[[Sequence @@ cur]] = myPrimeQ[seq[[i]]];
     i++;
     If[i > m^2, Goto[end]];
    ];
    numdir = 1 + Mod[numdir, 4];
    mult++;
   ];
   Label[end];
   ArrayPlot[mat, ColorFunction → "Rainbow"]
  ];
```

```
curpetOfPrimes[50]
```

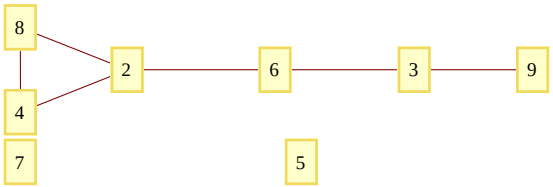


■ Elements of Graph Theory

Let V be a finite set, E be a set of 2-elements subsets of V.
The pair G = (V, E) is called a *graph* with the *vertex set* V=V(G) and the *edge set* E=E(G). We say that G *spans* V. Elements from V are called *vertices*, and from E *edges*.
Example: V={2, 3, 4, 5, 6, 7, 8, 9}, E consists of those {v,w} for which v divides w or w divides v and v ≠ w.
Thus E = { {2, 4}, {2, 6}, {2, 8}, {3, 6}, {3 ,9}, {4, 8} }.

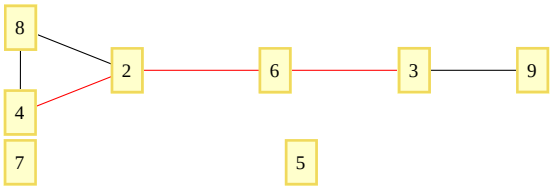
■ How to draw graphs in Mathematica

```
GraphPlot[{2 → 4, 2 → 6, 2 → 8, 3 → 6, 3 → 9, 4 → 8, 5 → 5, 7 → 7} (* the list of edges *),
 VertexLabeling → True (* write down the names of the vetrices *),
 SelfLoopStyle → False (* do not draw the selfloop for the vertex 5 *)
]
```



A *route* in a graph G = (V, E) *joining vertices* v and w of G is a sequence of vertices v_1 = v, … , v_n = w, such that $\{v_i, v_{i+1}\} \in$ E for all i.
Example: For the graph from previous example, draw a route joining 4 and 3

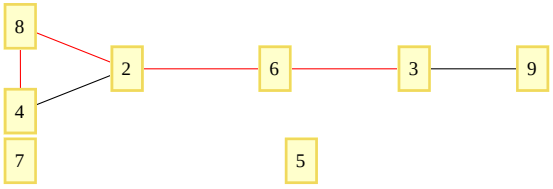
```
GraphPlot[{2 → 4, 2 → 6, 2 → 8, 3 → 6, 3 → 9, 4 → 8, 5 → 5, 7 → 7} (* the list of edges *),
 VertexLabeling → True (* write down the names of the vetrices *),
 SelfLoopStyle → False (* do not draw the selfloop for the vertex 5 *),
 EdgeRenderingFunction → (* #1 is the list of coordinates of consecutive
   points on the edge (usually, of ending points); #2 is the edge itself,
  namely, the pair of names of the vertices; #3 is the label *)
  (If[#2 == {2, 4} || #2 == {2, 6} || #2 == {3, 6}, {Red, Line[#1]}, {Black, Line[#1]}] &)
]
```



A route all whose edges $\{v_i, v_{i+1}\}$ are distinct is called a *path*.
Example: the route in previous picture is a path.

Another path is

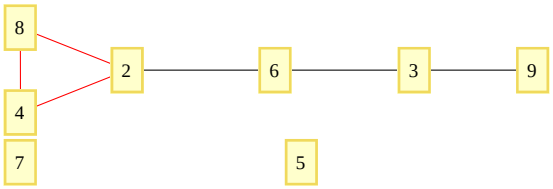
```
GraphPlot[{2 → 4, 2 → 6, 2 → 8, 3 → 6, 3 → 9, 4 → 8, 5 → 5, 7 → 7} (* the list of edges *),
 VertexLabeling → True (* write down the names of the vetrices *),
 SelfLoopStyle → False (* do not draw the selfloop for the vertex 5 *),
 EdgeRenderingFunction → (If[#2 == {4, 8} || #2 == {2, 8} || #2 == {2, 6} || #2 == {3, 6},
     {Red, Line[#1]}, {Black, Line[#1]}] &)
]
```



A *cycle* is a path joining v = w.

Example: our graph has just one cycle

```
GraphPlot[{2 → 4, 2 → 6, 2 → 8, 3 → 6, 3 → 9, 4 → 8, 5 → 5, 7 → 7} (* the list of edges *),
 VertexLabeling → True (* write down the names of the vetrices *),
 SelfLoopStyle → False (* do not draw the selfloop for the vertex 5 *),
 EdgeRenderingFunction →
  (If[#2 == {4, 8} || #2 == {2, 8} || #2 == {2, 4}, {Red, Line[#1]}, {Black, Line[#1]}] &)
]
```



A graph is called *connected* if any two its vertices can be joined by a route.

To use various graphs functions, we need to work with another representation of graphs.

Graph[

{{v_1, w_1}}, {{v_2, w_2}}, ...}, (* each {v_2, w_2} represents an edge by the numbers of its vertices *)

{ {{x_1, y_1}}, {{x_2, y_2}}, ...} (* coordinates of vertices in the plane *)

]

```
Needs["Combinatorica`"] (* we need this package to work with graphs *)
```

To convert the previous presentation to the later one, we need first to convert (a -> b) to {{a, b}}

```
(2 → 4)[[1]]
(2 → 4)[[2]]
```

```
2
```

```
4
```

```
Head[2 → 4]
```

```
Rule
```

```
Rule[2, 4]
```

```
2 → 4
```

- How one can change the Head

```
Apply[g, f[x, y]]
g @@ f[x, y]
```

```
g[x, y]
```

```
g[x, y]
```

For example,

```
Head[x + y]
Head[x y]
Times @@ (x + y)
```

```
Plus
```

```
Times
```

```
x y
```

```
List @@ (2 → 4)
```

```
{2, 4}
```

So, we can proceed as follows

```
edges = List[List @@ # - 1] & /@ {2 → 4, 2 → 6, 2 → 8, 3 → 6, 3 → 9, 4 → 8}
vertices =
 {{{0, 0}}, {{2, 0}}, {{-1, -1}}, {{1, -2}}, {{1, 0}}, {{-1, -2}}, {{-1, 1}}, {{3, 0}}}
```

```
{{{1, 3}}, {{1, 5}}, {{1, 7}}, {{2, 5}}, {{2, 8}}, {{3, 7}}}
```

```
{{{0, 0}}, {{2, 0}}, {{-1, -1}}, {{1, -2}}, {{1, 0}}, {{-1, -2}}, {{-1, 1}}, {{3, 0}}}
```

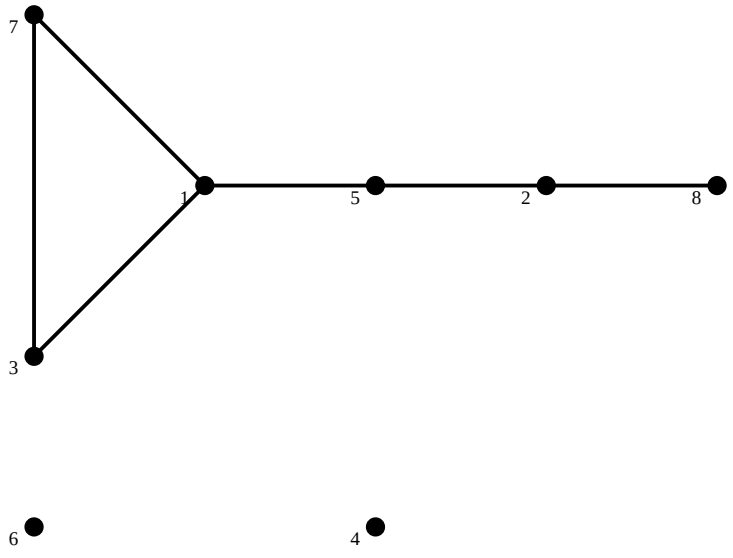
```
graph = Graph[edges, vertices]
```

```
- Graph:<6,8,Undirected>-
```

```
ConnectedQ[graph]
```

```
False
```

```
ShowGraph[graph, VertexNumber → True]
```

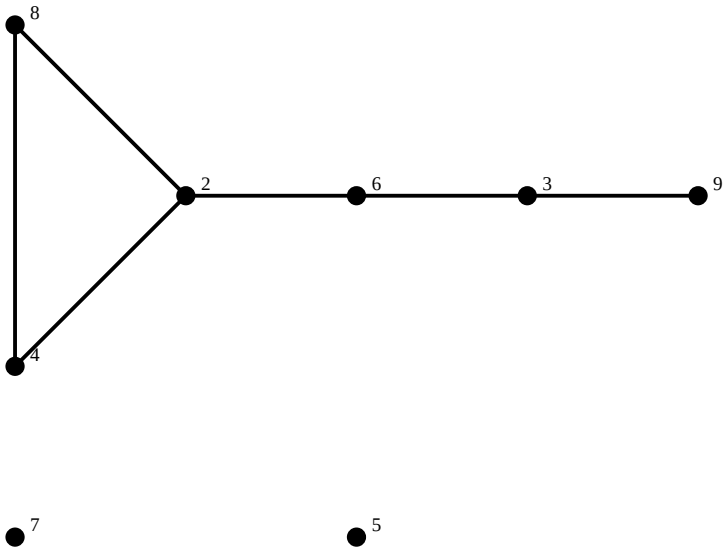


To get the vertices names as above, we'll use VertexLabel:
vertices = { {{x_1, y_1}, VertexLabel → label1}, {{x_2, y_2}, VertexLabel → label2}, ... }

```
vertices = MapIndexed[Append[#, VertexLabel → (#2[[1]] + 1)] &, vertices]
(* Append[expr,x] adds x to the end of expression expr *)
graph = Graph[edges, vertices]
ShowGraph[graph, VertexLabel → True]
```

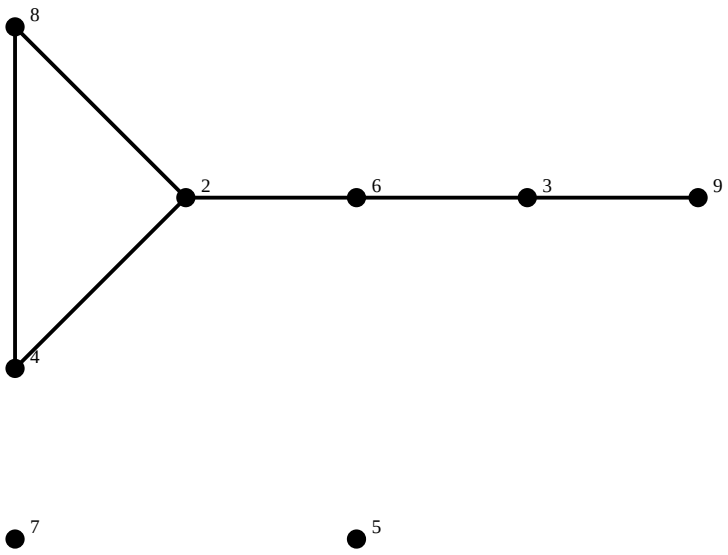
```
{{{0, 0}, VertexLabel → 2}, {{2, 0}, VertexLabel → 3},
 {{-1, -1}, VertexLabel → 4}, {{1, -2}, VertexLabel → 5}, {{1, 0}, VertexLabel → 6},
 {{-1, -2}, VertexLabel → 7}, {{-1, 1}, VertexLabel → 8}, {{3, 0}, VertexLabel → 9}}
```

```
- Graph:< 6,8,Undirected >-
```



Another way to set labels is to use the function SetVertexLabels[graph, {label1, label2, ...}]

```
vertices =
  {{{0, 0}}, {{2, 0}}, {{-1, -1}}, {{1, -2}}, {{1, 0}}, {{-1, -2}}, {{-1, 1}}, {{3, 0}}};
graph = Graph[edges, vertices];
graph = SetVertexLabels[graph, {2, 3, 4, 5, 6, 7, 8, 9}];
ShowGraph[graph, VertexLabel → True]
```



Maximal connected subgraph is called a *connected component*.
ConnectedComponents[graph] gives the vertices of graph partitioned into connected components.

```
ConnectedComponents[graph]
```

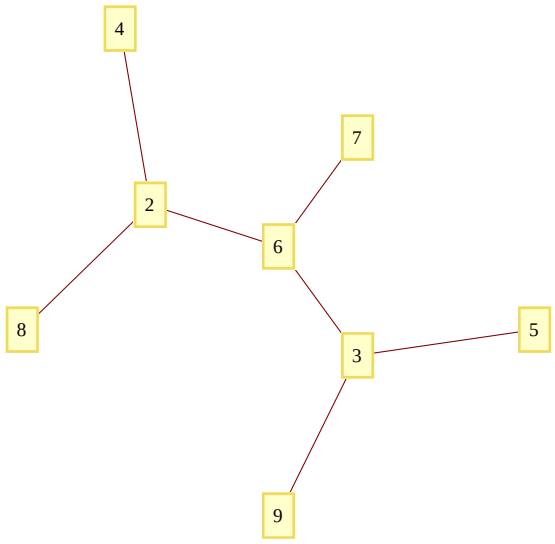
```
{{1, 2, 3, 5, 7, 8}, {4}, {6}}
```

A connected graph without cycles is called a *tree*.

```
TreeQ[graph]
```

```
False
```

```
GraphPlot[{2 → 4, 2 → 6, 2 → 8, 3 → 6, 3 → 9, 3 → 5, 6 → 7}, VertexLabeling → True]
```

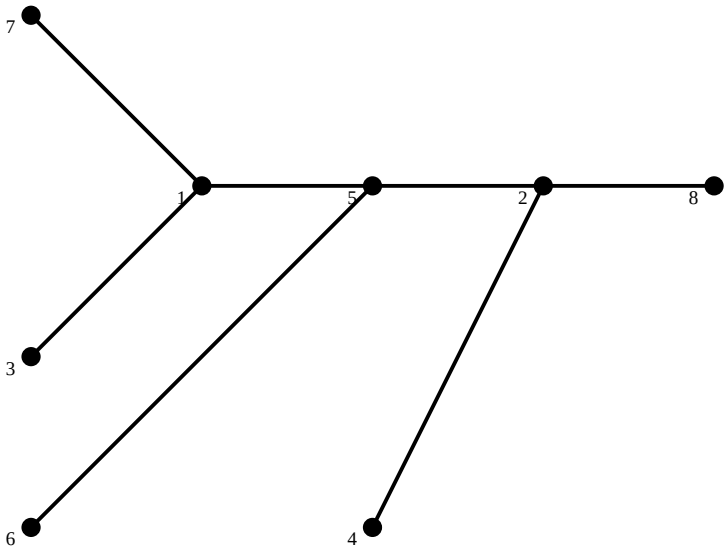


```
edges1 = List[List @@ # - 1] & /@ {2 → 4, 2 → 6, 2 → 8, 3 → 6, 3 → 9, 3 → 5, 6 → 7}
vertices1 =
 {{{0, 0}}, {{2, 0}}, {{-1, -1}}, {{1, -2}}, {{1, 0}}, {{-1, -2}}, {{-1, 1}}, {{3, 0}}}
graph1 = Graph[edges1, vertices1]
ShowGraph[graph1, VertexNumber → True]
TreeQ[graph1]
```

```
{{{1, 3}}, {{1, 5}}, {{1, 7}}, {{2, 5}}, {{2, 8}}, {{2, 4}}, {{5, 6}}}
```

```
{{{0, 0}}, {{2, 0}}, {{-1, -1}}, {{1, -2}}, {{1, 0}}, {{-1, -2}}, {{-1, 1}}, {{3, 0}}}
```

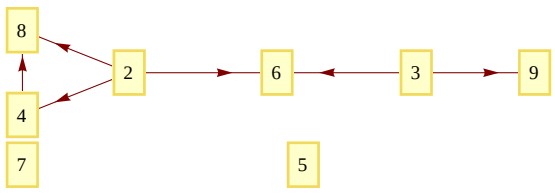
```
- Graph:<7,8,Undirected>-
```




```
True
```

Let V be a finite set, E be a set of ordered pairs of points from V.
The pair G = (V, E) is called an *oriented graph* with the *vertex set* V=V(G) and the *edge set* E=E(G).
Example: V={2, 3, 4, 5, 6, 7, 8, 9}, E consists of those {v, w} for which v divides w and v ≠ w.
Thus E = { {2, 4}, {2, 6}, {2, 8}, {3, 6}, {3, 9}, {4, 8} }.

```
GraphPlot[{2 → 4, 2 → 6, 2 → 8, 3 → 6, 3 → 9, 4 → 8, 5 → 5, 7 → 7} (* the list of edges *),
 VertexLabeling → True (* write down the names of the vetrices *),
 SelfLoopStyle → False (* do not draw the selfloop for the vertex 5 *),
 DirectedEdges → True (* draws the arrows instead lines *)
]
```



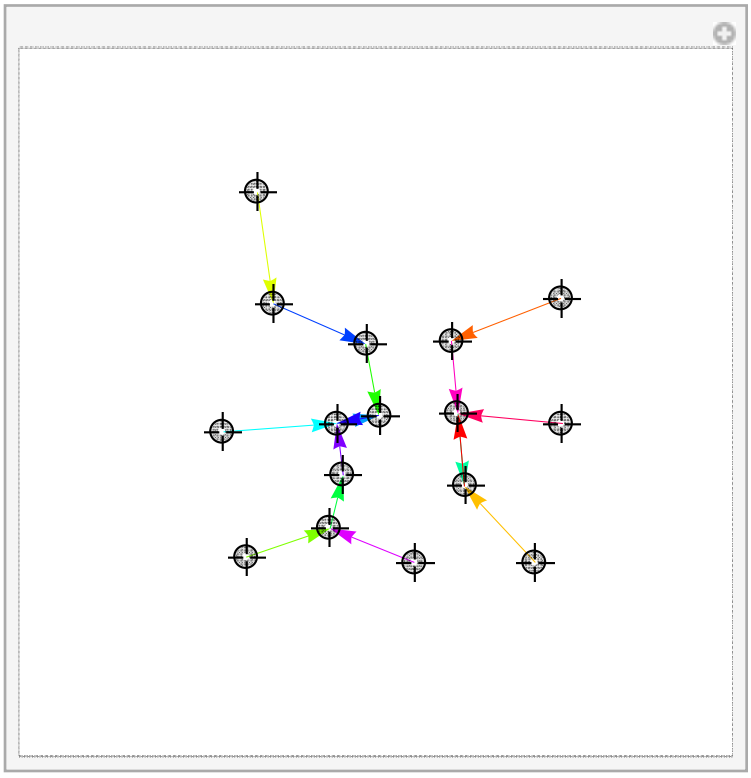
■ Nearest points

Given a finite M in $\mathbb{R}^n$, find a pair of points from M with the smallest Euclidean distance between them.
Construct an oriented graph G = (M, E) such that all edges going out from each its vertex come to the nearest points of M.

■ Algorithms (divide and conquer; Voronoi diagrams, see below)

■ *Mathematica* Implementation

```
Manipulate[
 Module[{rg, ar, lar},
  rg = Range[Length[pp]];
  ar = Flatten@MapThread[Outer[Composition[Arrow, List], {#1}, #2, 1] &,
     {pp, Nearest[pp[[Complement[rg, {#}]]], pp[[#]] & /@ rg}];
  lar = Length[ar];
  Graphics[MapIndexed[{Hue[#2[[1]]/lar], #} &, ar], PlotRange → {{-0.5, 1.5}, {-0.5, 1.5}}]
 ],
 {{pp, {{0, 0}, {1, 0}, {0, 1}}}, Locator, LocatorAutoCreate → True}]
```



■ Observations

1) A point can have at most six nearest neighbors in two dimensions, and at most 12 in three dimensions. This maximum number of nearest neighbors is the same as the maximum number of unit spheres that can be placed so as to touch a given one.

■ **Exercise**

Prove the Observation.

■ **Applications**

1) Air-traffic control problem: the two aircraft that are in the greatest danger of collision are the two closest ones.
2) Geography: the territoriality of species (Kolars, J. F. Nystuen, J. D. Geography: the study of location, culture, and environment).
3) A pair x, y of points from M such that (x, y) and (y, x) are edges of G is called *reciprocal* couples. In mathematical ecology, the ratio of reciproca1 pairs number to the total pairs number is used to detect whether members of a species tend to occur in isolated couples. Some statistics: if points are chosen in the plane according to a Poisson random point process, the expected fraction of reciprocal pairs is about 0.6215.

■ **Explanation of *Mathematica* objects used:**

1) `Nearest[{elem1, elem2, ...}, x]` gives the list of $elem_i$ to which x is nearest.

```
Nearest[{{0., 0.}, {1., 1.}, {2., 2.}}, {0.5, 0.5}]
```

```
{{0., 0.}, {1., 1.}}
```

2) Definitions of functions:

```
f[x_] := x^2;
f[3]
```

```
9
```

Recall that f is called the *head* of the function, and x is called the *argument;* _ is called *blank pattern* and stands for any *Mathematica* object; x_ is stand for a blank pattern with the name x.

Clear the definition:

```
ClearAll[f]
f[3]
```

```
f[3]
```

Definition of function without head

```
#^2 &
```

stands for argument; & tells that the previous expression is a function; for multiple arguments use #1, #2, ...

```
#^2 &[3]
```

```
9
```

3) Map[f, expr] applies f to each element on the first level in expr.

```
Map[f, {a, b, c}]
```

```
{f[a], f[b], f[c]}
```

```
Map[f, {a, b, {c, d}}]
```

```
{f[a], f[b], f[{c, d}]}
```

It's convenient to use in Map the functions without heads

```
Map[#^2 &, {a, b, c}]
```

```
{a^2, b^2, c^2}
```

Another form to write down Map is /@

```
#^2 & /@ {a, b, c}
```

```
{a^2, b^2, c^2}
```

4) MapIndexed[*f*, *expr*] applies *f* to the elements of *expr*, giving the part specification of each element as a second argument to *f*.

```
MapIndexed[f, {a, b, c}]
```

```
{f[a, {1}], f[b, {2}], f[c, {3}]}
```

```
MapIndexed[#1_(#2[[1]]) &, {a, b, c}]
```

```
{a_1, b_2, c_3}
```

5) MapThread[*f*, {{a_1, a_2, ...}, {b_1, b_2, ...}, ...}] gives {*f*[a_1, b_1, ...], *f*[a_2, b_2, ...], ...}.

To obtain MapIndexed by means of MapThread, one may proceed as follows :

```
MapThread[f[#1, {#2}] &, {{a, b, c}, Range[Length[{a, b, c}]]}]
```

```
{f[a, {1}], f[b, {2}], f[c, {3}]}
```

6) A few ways to write down functions:
One-argument function:

```
f[x]
f@x
x // f
```

```
f[x]
```

```
f[x]
```

```
f[x]
```

Two-arguments function:

```
f[x, y]
x ~ f ~ y
```

```
f[x, y]
```

```
f[x, y]
```

```
Flatten@{{a, b}, {c, d}}
```

```
{a, b, c, d}
```

```
(1 + x)^10 // Expand
```

```
1 + 10 x + 45 x^2 + 120 x^3 + 210 x^4 + 252 x^5 + 210 x^6 + 120 x^7 + 45 x^8 + 10 x^9 + x^10
```

```
{a, b} ~ Join ~ {c, d}
```

```
{a, b, c, d}
```

7) Complement[e_{all}, e_1, e_2, ...] gives the elements in e_{all} which are not in any of the e_i

```
Complement[{1, a2, b3, 4, 5}, {a2, 4, 7}]
```

```
{1, 5, b3}
```

8) `Outer[f, list1, list2, ...]` gives the generalized outer product of the $list_i$, forming all possible combinations of the lowest-level elements in each of them, and feeding them as arguments to f.

Almost Cartesian product

```
Outer[List, {a, b}, {x, y}]
```

```
{{{a, x}, {a, y}}, {{b, x}, {b, y}}}
```

If we delete unnecessary braces, we obtain Cartesian product

```
Flatten[Outer[List, {a, b}, {x, y}], 1]
```

```
{{a, x}, {a, y}, {b, x}, {b, y}}
```

Generalized Cartesian product

```
Outer[f, {a, b, c}, {x, y}]
```

```
{{f[a, x], f[a, y]}, {f[b, x], f[b, y]}, {f[c, x], f[c, y]}}
```

`Outer[f, list1, list2, ..., n]` treats as separate elements only sublists at level n in the $list_i$.

```
Outer[f, {a, b, c}, {{x1, x2}, {y1, y2}}]
```

```
{{{f[a, x1], f[a, x2]}, {f[a, y1], f[a, y2]}},
 {{f[b, x1], f[b, x2]}, {f[b, y1], f[b, y2]}}, {{f[c, x1], f[c, x2]}, {f[c, y1], f[c, y2]}}}
```

```
Outer[f, {a, b, c}, {{x1, x2}, {y1, y2}}, 1]
```

```
{{f[a, {x1, x2}], f[a, {y1, y2}]},
 {f[b, {x1, x2}], f[b, {y1, y2}]}, {f[c, {x1, x2}], f[c, {y1, y2}]}}
```

9) `Composition[f1, f2, f3, ...]` represents a composition of the functions f_1, f_2, f_3, ...

```
Composition[Arrow, List][p, q]
```

```
Arrow[{p, q}]
```

■ Description of *Mathematica* Implementation

```
Manipulate[
 Module[{rg, ar, lar},
  rg = Range[Length[pp]]; (* the list {1,2,...,n}, where n is the number of locators *)
  ar = Flatten@MapThread[ (* gives {Arrow[{locator1, its nearest11}],
        Arrow[{locator1, its nearest12}],...,Arrow[{locator2, its nearest21}],...} *)
     Outer[(*  gives {Arrow[{locatori, its nearesti1}],
         Arrow[{locatori, its nearesti2}],...} *)
       Composition[Arrow, List],
       {#1}, (* each locator *)
       #2, (* the list of the nearest locators *)
       1
      ] &,
     {
      pp,
      Nearest[
         pp⟦Complement[rg, {#}]⟧,
         pp⟦#⟧
        ] &/@ rg (* for each i ∈ {1,2, ... , n} we
         calculate the locator nearest to p⟦i⟧ among the remaining ones
         Example: let pp = {p1,p2,p3}, then rg = {1,2,3}
           Nearest[ {p1,p2,p3}⟦Complement[{1,2,3},{#}]⟧, {p1,p2,p3}⟦#⟧ ] &/@ {1,2,3}
        gives
            {Nearest[{p2,p3},p1],Nearest[{p1,p3},p2],Nearest[{p1,p2},p3]}
      *)
     }
    ];
  lar = Length[ar]; (* the number of arrows *)
  Graphics[(* draw all arrows in different colors using Hue *)
   MapIndexed[{Hue[#2⟦1⟧/lar], #} &, ar], PlotRange → {{-0.5, 1.5}, {-0.5, 1.5}}
  ]
 ],
 {{pp, {{0, 0}, {1, 0}, {0, 1}}}, Locator, LocatorAutoCreate → True}]
```

Find Shortest Tour (Travelling salesman problem)

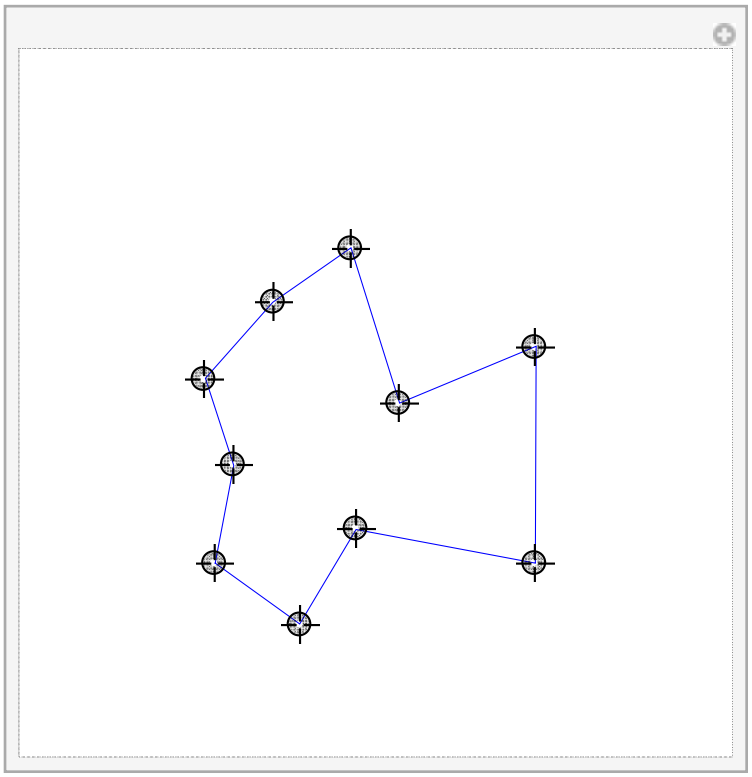
Given M $\subset \mathbb{R}^n$, construct the shortest tour that visits all the points from M once.

Algorithms (brute force search - all permutations; inclusion–exclusion;branch-and-bound; linear programming; cutting-plane method - Dantzig G. , Fulkerson Rıo, and Johnson S. ; heuristic and approximation algorithms).

http://en.wikipedia.org/wiki/Travelling_salesman_problem

***Mathematica* Implementation**

```
Manipulate[
 Module[{ord},
  ord = FindShortestTour[pp] // Last;
  ord = ord ~Join~ {First[ord]};
  Graphics[{Blue, Line[ pp[[ord]]]}, PlotRange → {{-0.5, 1.5}, {-0.5, 1.5}}]
 ], {{pp, {{0, 0}, {1, 0}, {0, 1}}}, Locator, LocatorAutoCreate → True}]
```



Remark. For small numbers of points, `FindShortestTour` will usually find the shortest possible tour. For larger numbers of points it will normally find a tour whose length is at least close to the minimum.

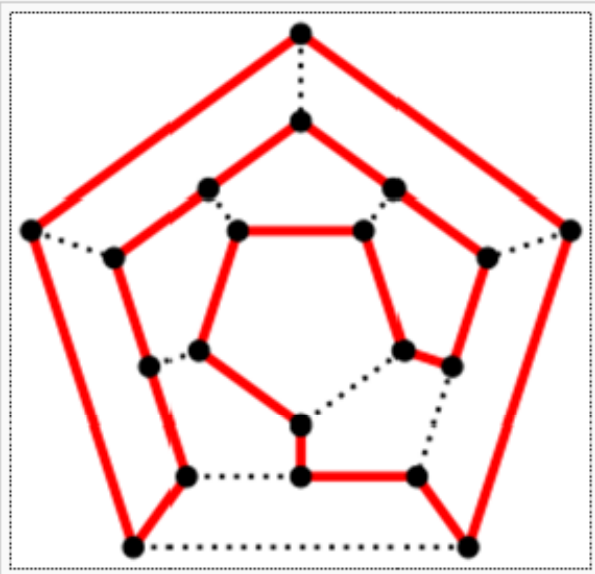
Applications

1) Travelling salesman (tourist, etc.) problem

2) The Tube Challenge (the accepted name for the Guinness World Record attempt to visit all of the London Underground stations in the fastest time possible, of which there are currently 270; the length of time required to visit all stations on the network is currently at around seventeen hours - only slightly shorter than the daily operating hours of the system; completing the challenge in a single day is therefore difficult; the current official Guinness World Record stands at 16 hours, 44 minutes and 16 seconds which was achieved by Martin Hazel, Andi James and Steve Wilson on the 14th December 2009).

Related problems

1) Given a complete weighted graph (where the vertices would represent the cities, the edges would represent the roads, and the weights would be the cost or distance of that road), find a Hamiltonian cycle (i.e, a path in an undirected graph which visits each vertex exactly once) with the least weight.



A Hamiltonian cycle in a dodecahedron.

2) Bottleneck travelling salesman problem (bottleneck TSP): find a Hamiltonian cycle in a weighted graph with the minimal weight of the weightiest edge (in printed circuit manufacturing: scheduling of a route of the drill machine to drill holes in a printed circuit board.).

3) "Travelling politician problem": given "states" that have (one or more) "cities", politician has to visit exactly one "city" from each "state" (Behzad and Modarres demonstrated that this problem can be transformed into a standard travelling salesman problem with the same number of cities, but a modified distance function).

- **Explanation of *Mathematica* objects used:**

1) `FindShortestTour[{e1, e2, …}]` attempts to find an ordering of the e_i that minimizes the total distance on a tour that visits all the e_i once.

```
FindShortestTour[{{0., 0.}, {1., 0.}, {0., 1.}, {0.3, 0.3}}]
 (* the answer: {the length of the shortest tour, the order}*)
```

```
{3.52315, {1, 2, 4, 3}}
```

2) `Last[expr]` gives the last element in *expr*.

```
Last[{a, b, c, d}]
{a, b, c, d} // Last
```

```
d
```

```
d
```

■ Description of *Mathematica* Implementation

```
Manipulate[
 Module[{ord},
  ord = FindShortestTour[ pp] // Last; (* culculates the order of the optimal path *)
  ord = ord ~Join~ {First[ord]}; (* adds the number of the first point *)
  Graphics[{Blue, Line[ pp[[ord]]]}, PlotRange → {{-0.5, 1.5}, {-0.5, 1.5}}]
   (* draws the tour passing through given locators pp *)
 ], {{pp, {{0, 0}, {1, 0}, {0, 1}}}, Locator, LocatorAutoCreate → True}]
```

◄ | ►

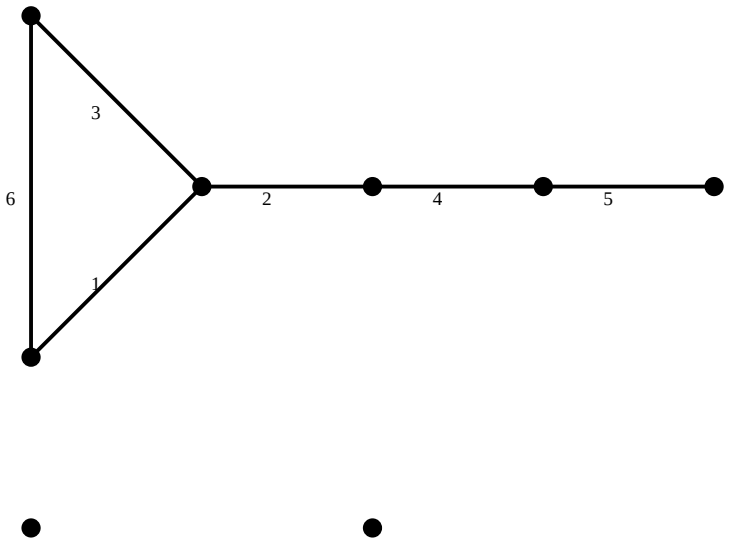
■ Elements of Graph Theory

Let G = (V, E) be a graph and $\omega : E \to \mathbb{R}_+$ a nonnegative real-valued function. Such ω is called a *weight function*. The pair (G, ω) is called a *weighted graph*. Sometimes we'll write G = (V, E, ω).
To setup a weighted graph, one can use EdgeWeight.

```
edges = {{{1, 3}, EdgeWeight → 0.1}, {{1, 5}, EdgeWeight → 3.4}, {{1, 7}, EdgeWeight → 81},
  {{2, 5}, EdgeWeight → 11}, {{2, 8}, EdgeWeight → 2.7}, {{3, 7}, EdgeWeight → 0.6}}
vertices = {{{0, 0}}, {{2, 0}}, {{-1, -1}}, {{1, -2}},
  {{1, 0}}, {{-1, -2}}, {{-1, 1}}, {{3, 0}}}
graph = Graph[edges, vertices];
ShowGraph[graph, EdgeLabel → True]
```

```
{{{1, 3}, EdgeWeight → 0.1}, {{1, 5}, EdgeWeight → 3.4}, {{1, 7}, EdgeWeight → 81},
 {{2, 5}, EdgeWeight → 11}, {{2, 8}, EdgeWeight → 2.7}, {{3, 7}, EdgeWeight → 0.6}}
```

```
{{{0, 0}}, {{2, 0}}, {{-1, -1}}, {{1, -2}}, {{1, 0}}, {{-1, -2}}, {{-1, 1}}, {{3, 0}}}
```



To get the list of weights, one can use GetEdgeWeights[graph]

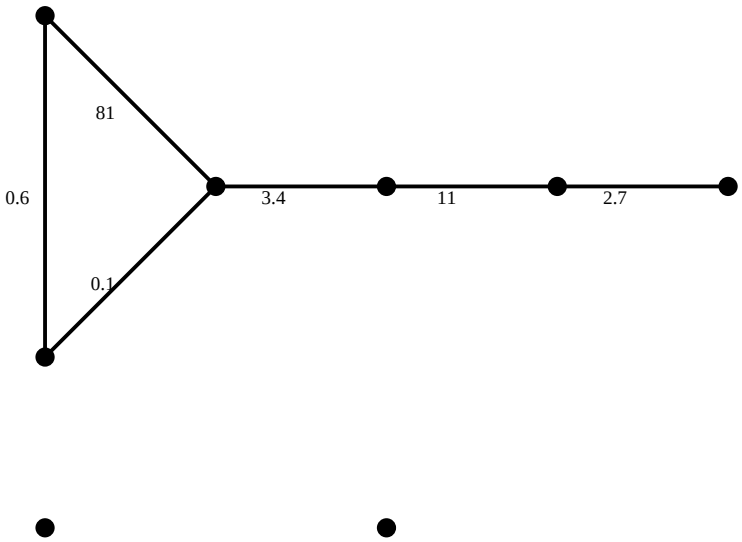
```
weights = GetEdgeWeights[graph]
```

```
{0.1, 3.4, 81, 11, 2.7, 0.6}
```

To visualize the weights, one can assign them as edges labels

```
graph = SetEdgeLabels[graph, weights]
ShowGraph[graph, EdgeLabel → True]
```

```
- Graph:<6,8,Undirected>-
```

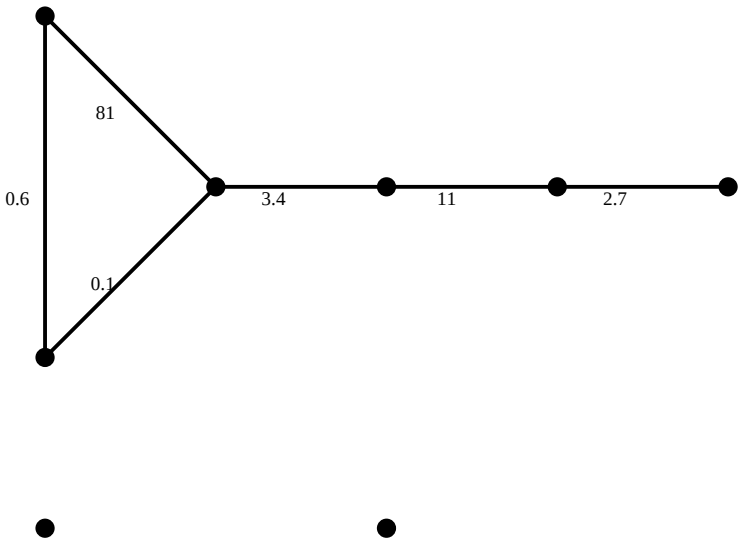


Another way to set the edges weights is to use SetEdgeWeights[graph, {weight1, weight2, ...}]

```
edges = {{{1, 3}}, {{1, 5}}, {{1, 7}}, {{2, 5}}, {{2, 8}}, {{3, 7}}}
vertices =
 {{{0, 0}}, {{2, 0}}, {{-1, -1}}, {{1, -2}}, {{1, 0}}, {{-1, -2}}, {{-1, 1}}, {{3, 0}}}
graph = Graph[edges, vertices];
weights = {0.1, 3.4, 81, 11, 2.7, 0.6};
graph = SetEdgeWeights[graph, weights];
graph = SetEdgeLabels[graph, weights];
ShowGraph[graph, EdgeLabel → True]
```

```
{{{1, 3}}, {{1, 5}}, {{1, 7}}, {{2, 5}}, {{2, 8}}, {{3, 7}}}
```

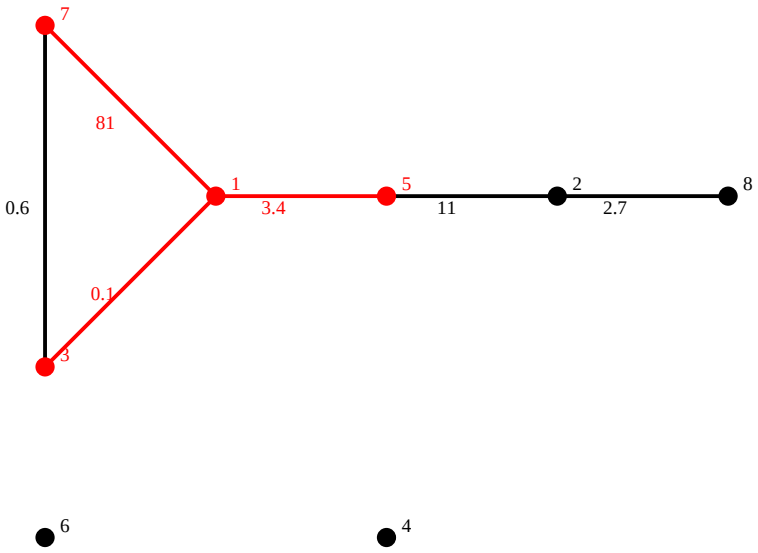
```
{{{0, 0}}, {{2, 0}}, {{-1, -1}}, {{1, -2}}, {{1, 0}}, {{-1, -2}}, {{-1, 1}}, {{3, 0}}}
```



For a graph G = (V, E) consider a pair (U, F), where U ⊂ V and F ⊂ E such that for any {v, w} ∈ F the both v and w belongs to U. Such H = (U, F) is called a *subgraph* of G. We write this as H ⊂ G. If G = (V, E, ω) is a weighted graph, by its *subgaph* we mean (U, F, ω), where H = (U, F) ⊂ G and by ω we denote the restriction of ω to F.

How to emphasis a subgraph?

```
edges = {{{1, 3}, EdgeColor → Red}, {{1, 5}, EdgeColor → Red},
    {{1, 7}, EdgeColor → Red}, {{2, 5}}, {{2, 8}}, {{3, 7}}};
vertices = {{{0, 0}, VertexColor → Red}, {{2, 0}}, {{-1, -1}, VertexColor → Red}, {{1, -2}},
    {{1, 0}, VertexColor → Red}, {{-1, -2}}, {{-1, 1}, VertexColor → Red}, {{3, 0}}};
graph = Graph[edges, vertices];
weights = {0.1, 3.4, 81, 11, 2.7, 0.6};
graph = SetEdgeWeights[graph, weights];
graph =
  SetEdgeLabels[graph, {Style[0.1, Red], Style[3.4, Red], Style[81, Red], 11, 2.7, 0.6}];
graph = SetVertexLabels[graph, {Style[1, Red], 2,
    Style[3, Red], 4, Style[5, Red], 6, Style[7, Red], 8}];
ShowGraph[graph, EdgeLabel → True, VertexLabel → True]
```



For any subgraph H of G the weight (H) is the value $\Sigma_{e\in E(H)}\ \omega(e)$.

How to calculate the weight of a weighted graph?

```
edges = {{{1, 3}, EdgeWeight → 0.1}, {{1, 5}, EdgeWeight → 3.4}, {{1, 7}, EdgeWeight → 81},
  {{2, 5}, EdgeWeight → 11}, {{2, 8}, EdgeWeight → 2.7}, {{3, 7}, EdgeWeight → 0.6}}
vertices = {{{0, 0}}, {{2, 0}}, {{-1, -1}}, {{1, -2}},
  {{1, 0}}, {{-1, -2}}, {{-1, 1}}, {{3, 0}}}
graph = Graph[edges, vertices];
weights = GetEdgeWeights[graph]
weightOfGraph =
 Plus @@ weights (* we change the head List to the head Plus to get the sum *)
```

```
{{{1, 3}, EdgeWeight → 0.1}, {{1, 5}, EdgeWeight → 3.4}, {{1, 7}, EdgeWeight → 81},
 {{2, 5}, EdgeWeight → 11}, {{2, 8}, EdgeWeight → 2.7}, {{3, 7}, EdgeWeight → 0.6}}
```

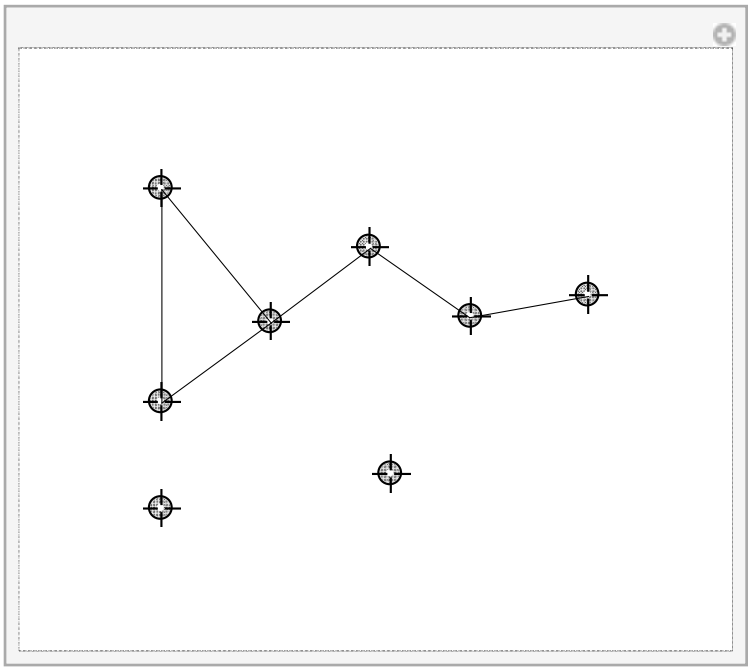
```
{{{0, 0}}, {{2, 0}}, {{-1, -1}}, {{1, -2}}, {{1, 0}}, {{-1, -2}}, {{-1, 1}}, {{3, 0}}}
```

```
{0.1, 3.4, 81, 11, 2.7, 0.6}
```

```
98.8
```

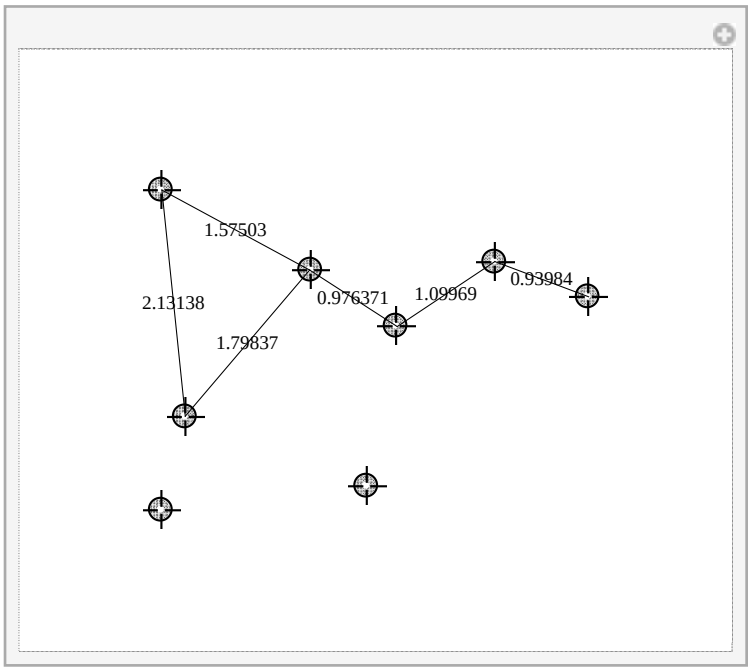
How one can animate a graph?

```
Manipulate[
 edges = {{{1, 3}}, {{1, 5}}, {{1, 7}}, {{2, 5}}, {{2, 8}}, {{3, 7}}};
 vertices = {#} & /@ pp;
 graph = Graph[edges, vertices];
 MyShowGraph[graph, PlotRange → {{-2, 4}, {-3, 2}}],
 {{pp, {{0, 0}, {2, 0}, {-1, -1}, {1, -2}, {1, 0}, {-1, -2}, {-1, 1}, {3, 0}}}, Locator},
 SaveDefinitions → True, Initialization :> (AbortProtect[Needs["Combinatorica`"]];
   MyShowGraph[g_, opts : OptionsPattern[Graphics]] :=
    Graphics[Line[Vertices[g][[#]] & /@ Edges[g], opts])
]
```



How to calculate and visualize Euclidean weights?

```
MyShowWeightedGraph[g_, opts : OptionsPattern[Graphics]] :=
  Graphics[(Line[Vertices[g][[#]] & /@ Edges[g]) ~Join~ MapThread[
     Text[#2, Mean @@ {Vertices[g][[#1]]}] &, {Edges[g], GetEdgeWeights[g]}], opts];
Manipulate[Module[{edges, vertices, graph, weights},
  edges = {{{1, 3}}, {{1, 5}}, {{1, 7}}, {{2, 5}}, {{2, 8}}, {{3, 7}}};
  vertices = {#} & /@ pp;
  graph = Graph[edges, vertices];
  graph = SetEdgeWeights[graph, WeightingFunction -> Euclidean];
  weights = GetEdgeWeights[graph];
  MyShowWeightedGraph[graph, PlotRange → {{-2, 4}, {-3, 2}}]
 ],
 {{pp, {{0, 0}, {2, 0}, {-1, -1}, {1, -2}, {1, 0}, {-1, -2}, {-1, 1}, {3, 0}}}, Locator},
 SaveDefinitions → True, Initialization :↦ (AbortProtect[Needs["Combinatorica`"]];
   MyShowWeightedGraph[g_, opts : OptionsPattern[Graphics]] :=
    Graphics[(Line[Vertices[g][[#]] & /@ Edges[g]) ~Join~ MapThread[
       Text[#2, Mean @@ {Vertices[g][[#1]]}] &, {Edges[g], GetEdgeWeights[g]}], opts])
]
```



Minimum Spanning Tree (MST)

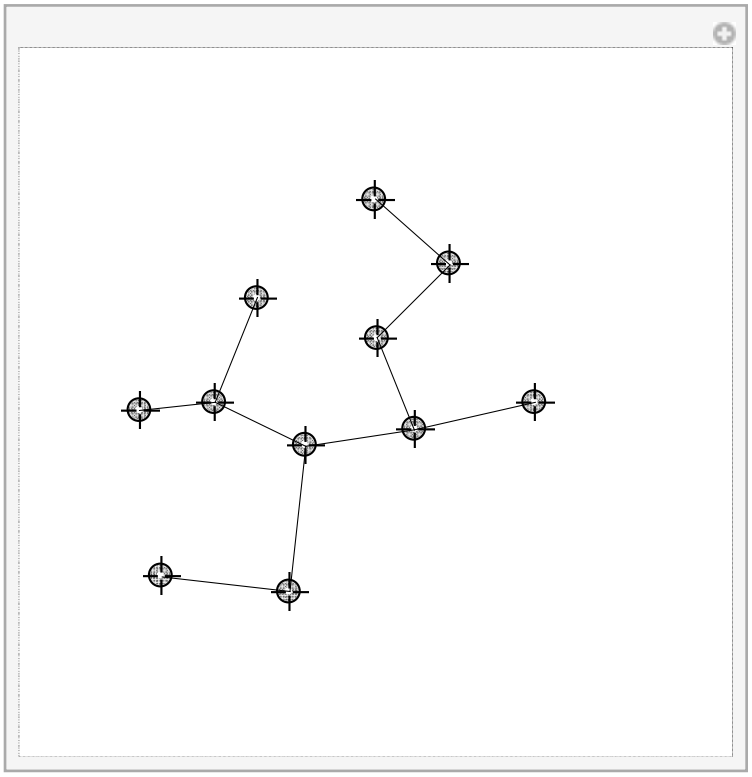
If (G, ω) is a connected weighted graph, then any subtree Δ of G such that V (Δ) = V (G) is called a spanning tree.

Given a weighted graph G, each its spanning tree of the least possible weight is called a *minimum spanning tree* for G.
If G = (M, E, ρ), where (M, ρ) is a metric space, and E consists of all possible edges (such G is called *complete graph*), each minimum spanning tree in G is called a *minimum spanning tree on the metric space* (M, ρ).
If M $\subset \mathbb{R}^n$ and ρ is Euclidean distance, then each minimum spanning tree on (M, ρ) is called a *Euclidean minimum spanning tree* on M.

- **Algorithms (Otakar Boruvka - the first, Prim, Kruskal, Bernard Chazelle - the fastest)**
  http://en.wikipedia.org/wiki/Minimum_spanning_tree

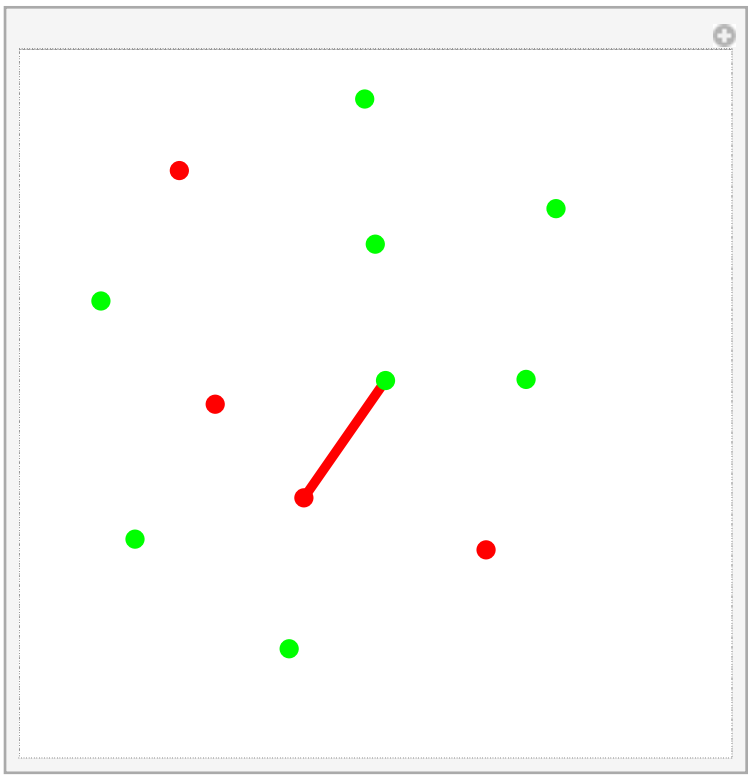
- ***Mathematica* Implementation**

```
Manipulate[
 Module[{vv, mst, g},
  g = SetEdgeWeights[
    ChangeVertices[CompleteGraph[Length[pts]], pts], WeightingFunction → Euclidean];
  mst = ChangeVertices[MinimumSpanningTree[g], pts];
  MyShowGraph[mst, PlotRange → {{-2, 2}, {-2, 2}}]
 ],
 {{pts, {{1, 0}, {0, 1}, {-1, 0}}}, Locator, LocatorAutoCreate → True},
 SaveDefinitions → True, Initialization :> (AbortProtect[Needs["Combinatorica`"]];
   MyShowGraph[g_, opts : OptionsPattern[Graphics]] :=
    Graphics[Line[Vertices[g][[#]] & /@ Edges[g], opts])]
```



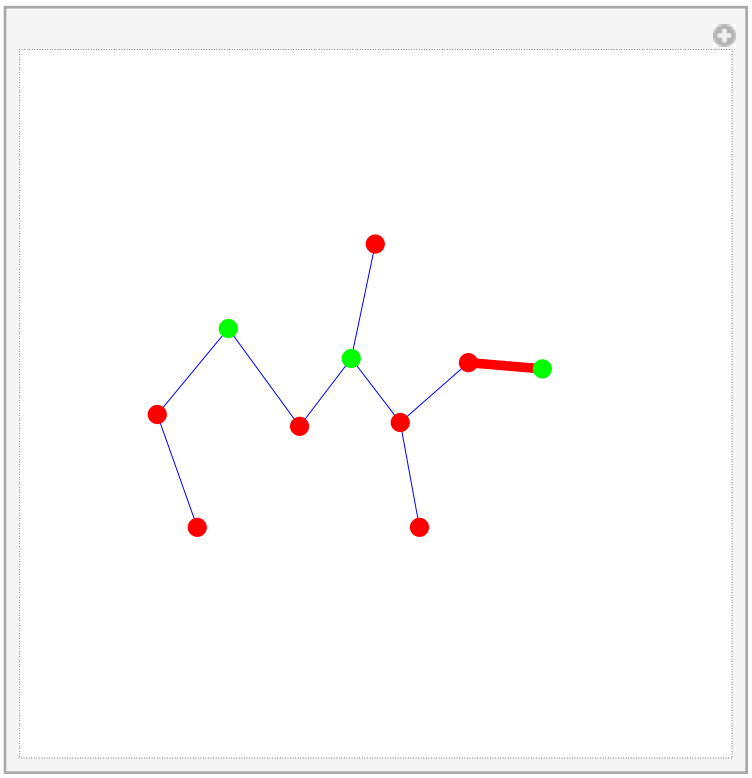
- **Observations**

1) Partition a set M $\subset \mathbb{R}^n$ into two subsets, calculate and visualize the distance between these subsets.

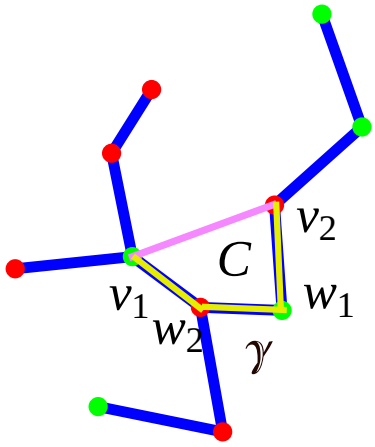


To manipulate with the implementation, press Ctrl key to mark red the point under mouse cursor, and Shift key to unmark. To add/remove a point use Alt+LeftClick as usual.

2) Add MST.



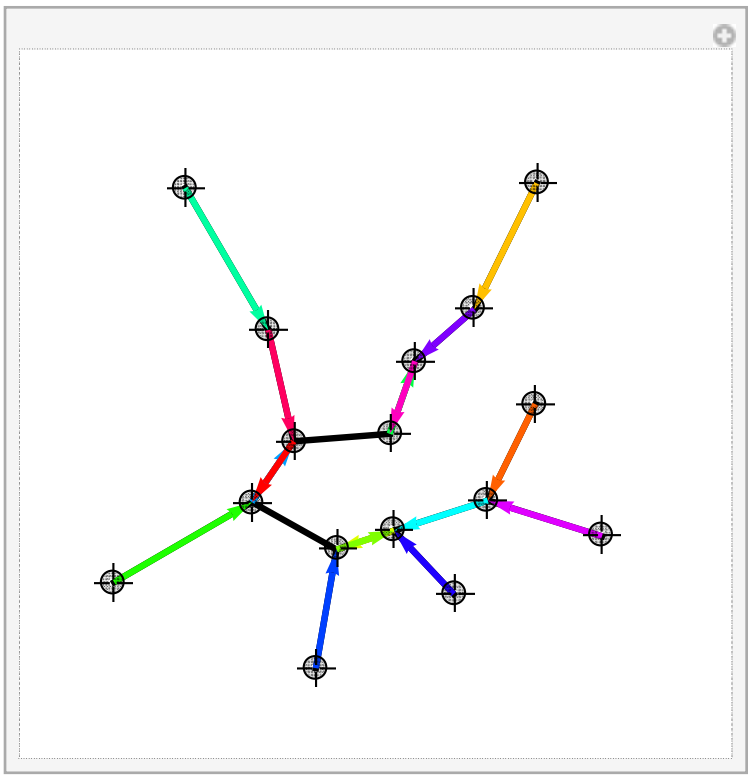
Observation. Each segment realizing the distance between elements of the partition belongs to MST.
Remark. It is not true. Suppose that we are given by regular triangle. Then we must draw all its sides. But no one MST contains all of them.
Correction. The Observation holds under assumption that all distances are different. For general situation, these segments belong to the union of all possible MST.
Proof. Suppose that all distances between points in M are different.
Partition the set M into M_1 and M_2 (red and green) and let dist[M_1, M_2] = dist[v_1, v_2], where $v_i \in M_i$.
Suppose that $\{v_1, v_2\}$ is not an edge of MST.
Join v_1 and v_2 by the unique path γ in MST.
$\gamma \bigcup \{v_1, v_2\}$ forms a cycle C in MST $\bigcup \{v_1, v_2\}$.
The cycle C contains an edge $\{w_1, w_2\}$ of MST such that $w_i \in M_i$.
The segment $[w_1, w_2]$ is shorter than $[v_1, v_2]$ because otherwise the edge $\{w_1, w_2\}$ can be replaced by $\{v_1, v_2\}$ to obtain a spanning tree which is shorter than MST.
Thus dist[M_1, M_2] ≤ dist[w_1, w_2] < dist[v_1, v_2] = dist[M_1, M_2], a contradiction.



Corollary. If all distances in M are different, then MST is unique.

Exercise. Prove this corollary.

3) Draw the both MST and Nearest points graph on the same picture.



Observation. Nearest points graph belongs to MST.
Remark. It is not true. Suppose that all edges weights in a complete graph G are the same. Then the nearest point graph equals G. However, no one subtree contains G.
Correction. The Observation holds under assumption that all the weights are different. For general weighted graph G, its nearest point graph belongs to the union of all possible MST in G.

Exercise. Prove the corrected form of the above Observation.

- **Related problems**

1) Construct a *k-minimum spanning tree* (*k-MST*) which is the tree that spans some subset of k vertices in the graph with minimum weight.
2) Construct a *set of k-smallest spanning trees*, i.e., a subset of k spanning trees (out of all possible spanning trees) such that no spanning tree outside the subset has smaller weight.
3) Construct a *Euclidean minimum spanning tre*e, i.e., a spanning tree of a graph with edge weights corresponding to the Euclidean distance between vertices which are points in the plane (or space).
4) The *rectilinear norm* of a vector is the sum of absolute values of all its coordinates. The distance function generated by such a norm is called *rectilinear distance*. Construct a *rectilinear minimum spanning tree*, i.e., a spanning tree of a graph with edge weights corresponding to the rectilinear distance between vertices which are points in the plane (or space).
5) For directed graphs, the minimum spanning tree problem is called the *Arborescence problem*.
6) Construct a *maximum spanning tree*, i.e., a spanning tree with weight greater than or equal to the weight of every other spanning tree.

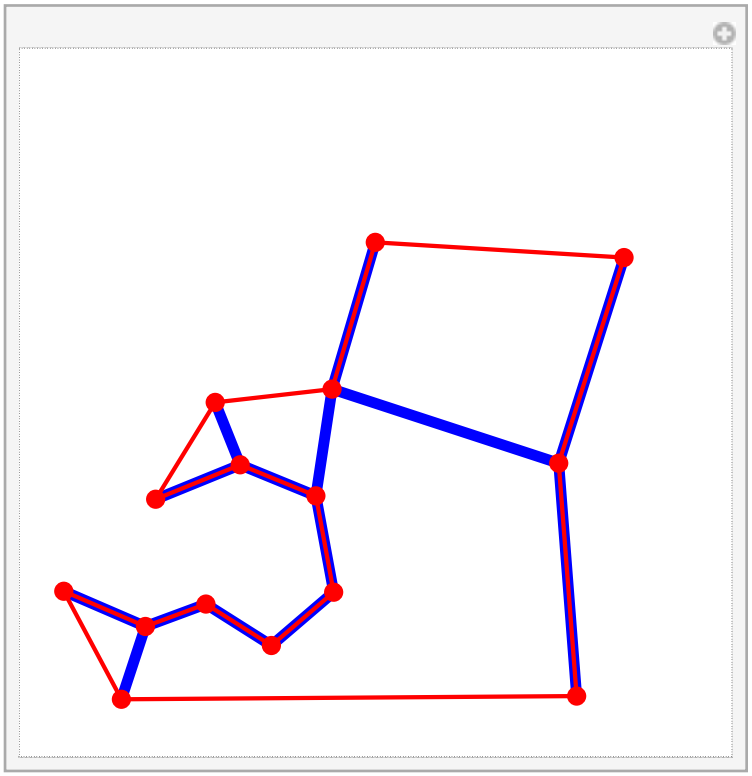
- **Applications**

1) Phone network design. Suppose that you have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost.
2) MST for finite $M \subset \mathbb{R}^n$ can be used to approximately solve the traveling salesman problem for M.

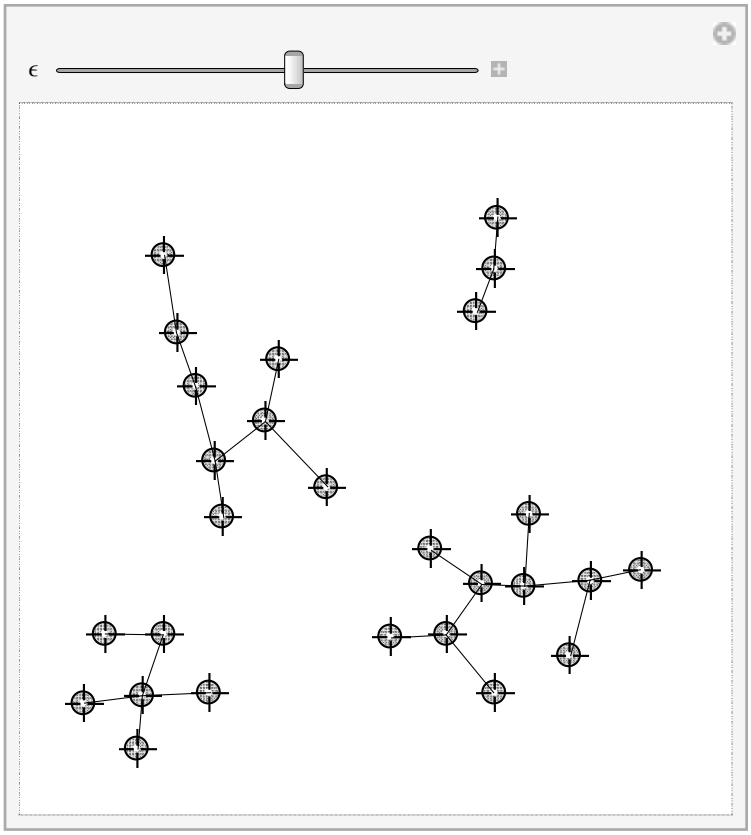
Observation. Each Euclidean MST is a plane graph, i.e., it does not have self-intersections.

Proof. It follows from the fact that the sum of diagonals of a convex 4-gon in the plane is greater than the sum of any pair of opposite sides of the 4-gon.

The approximate solution is the tour around MST.



3) Clustering: given a set of objects and distances between them (increasing distance between some objects corresponds to decreasing their similarity). Goal: group objects into clusters, where each cluster is a set of similar objects.

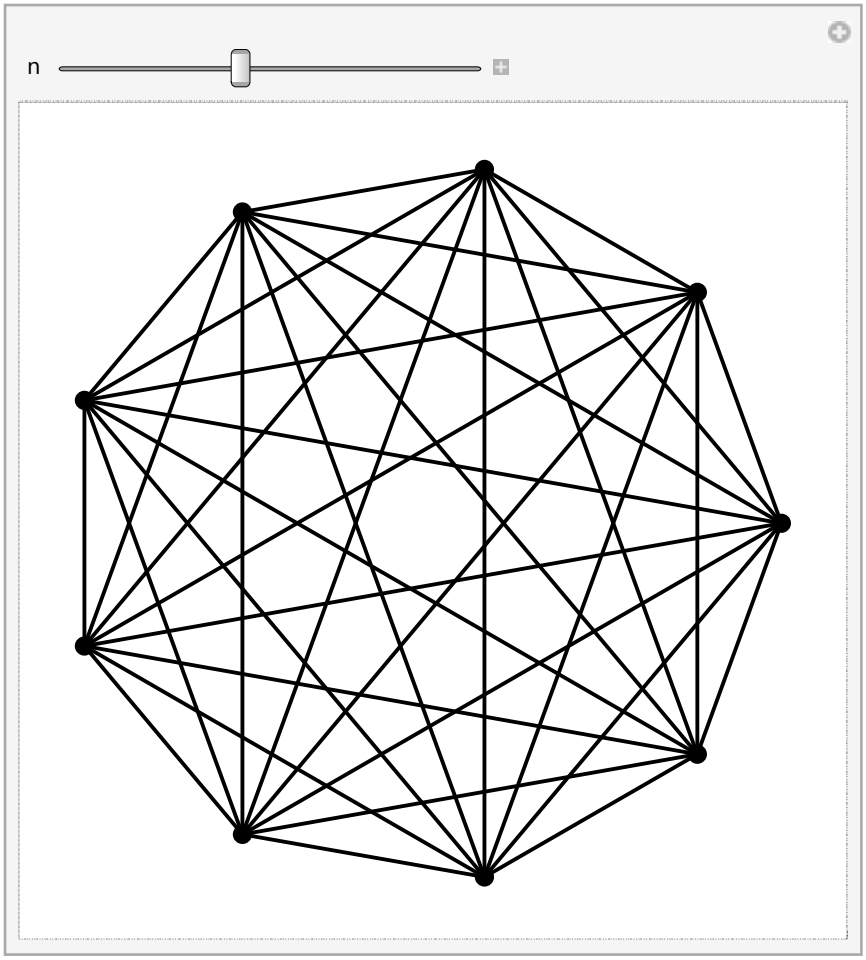


Clustering can be used for images recognition.

- **Explanation of *Mathematica* objects used:**

CompleteGraph[n] creates a complete graph on n vertices.

```
Manipulate[ShowGraph[CompleteGraph[n]], {{n, 3}, 1, 20, 1}]
```



MinimumSpanningTree[g] finds a minimum spanning tree of the graph g.

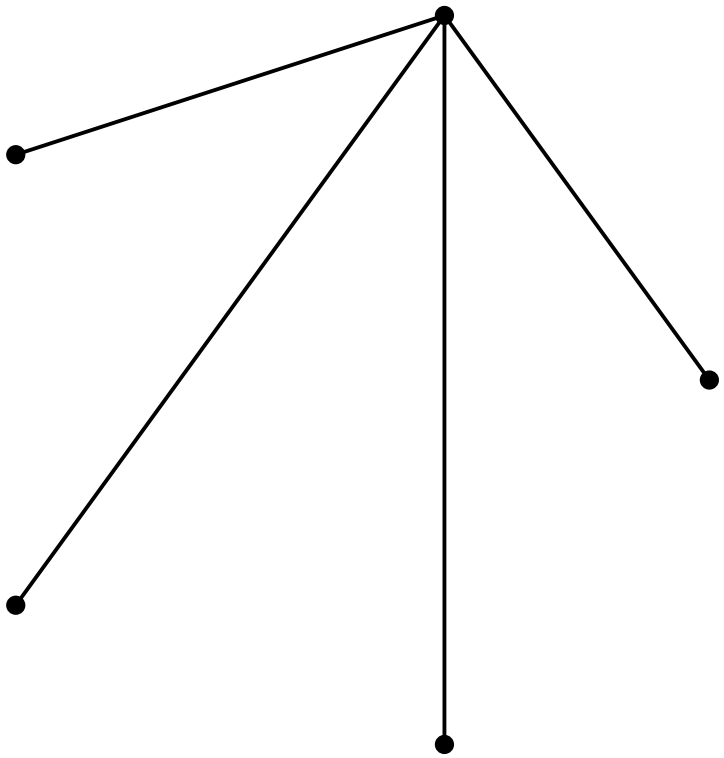
```
GetEdgeWeights[CompleteGraph[5]]
```

```
{1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
```

```
mst = MinimumSpanningTree[CompleteGraph[5]]
```

```
- Graph:<4,5,Undirected>-
```

```
ShowGraph[mst]
```



ChangeVertices[g, v] replaces the vertices of graph g with the vertices in the given list v.
v can have the form {{ x_1, y_1}, {x_2, y_2}, ... }, where { x_1, y_1}, { x_2, y_2}, ... are coordinates of points

```
Vertices[CompleteGraph[5]]
graph = ChangeVertices[CompleteGraph[5], {{0, 0}, {0, 1}, {0, 2}, {0, 3}, {0, 5}}]
Vertices[graph]
```

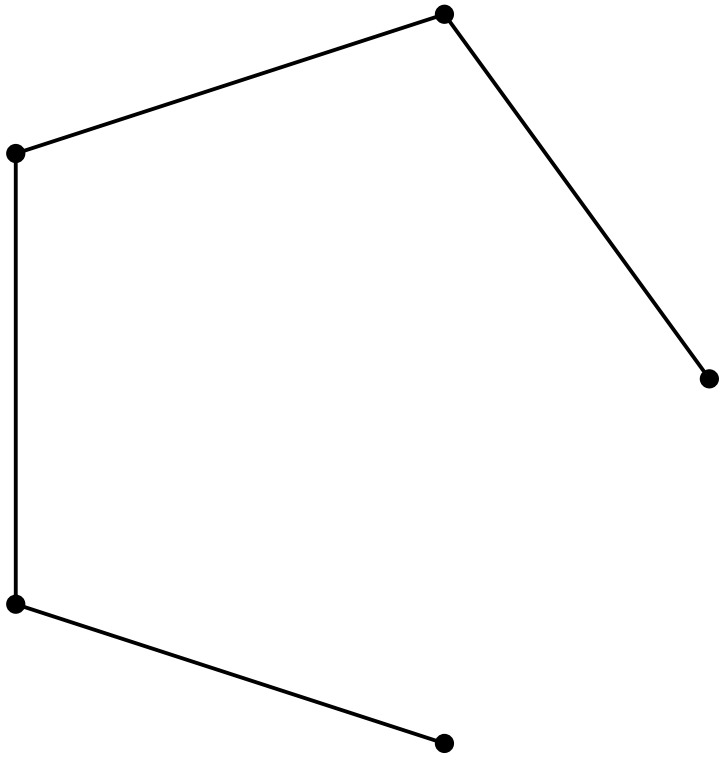
```
{{0.309017, 0.951057}, {-0.809017, 0.587785},
 {-0.809017, -0.587785}, {0.309017, -0.951057}, {1., 0}}
```

```
- Graph:<10,5,Undirected>-
```

```
{{0, 0}, {0, 1}, {0, 2}, {0, 3}, {0, 5}}
```

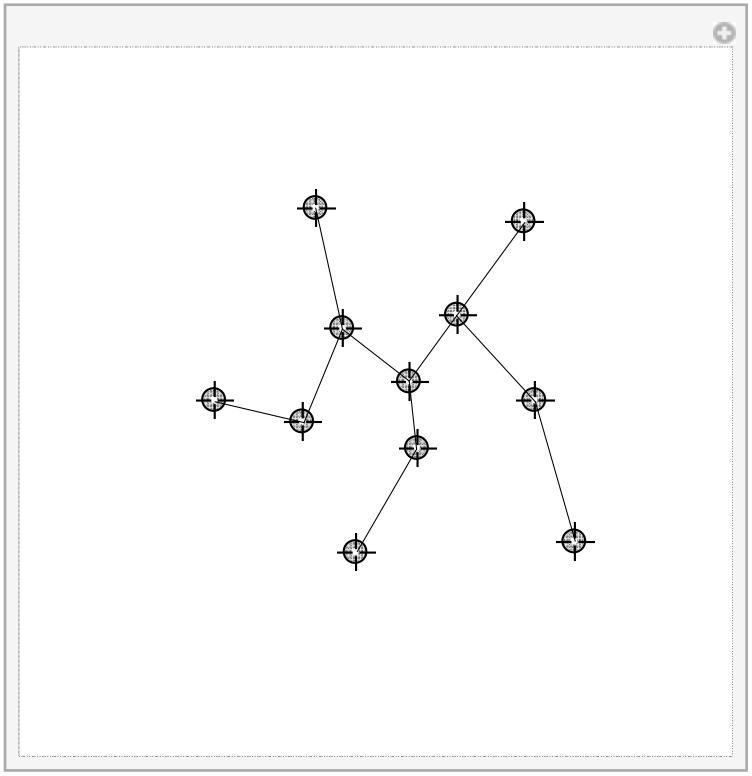
How to get MST of Euclidean complete graph on regular n-gon?

```
graph = CompleteGraph[5];
graph = SetEdgeWeights[graph, WeightingFunction → Euclidean];
ShowGraph[MinimumSpanningTree[graph]]
```



■ Description of *Mathematica* Implementation

```
Manipulate[
 Module[{vv, mst, g},
  g = SetEdgeWeights[ChangeVertices[
      (* replaces the vertices of graph g with coordinates of locators *)
      CompleteGraph[Length[pts]], (* Complete graph with Length[pts] vertices *)
      pts (* Locators *)
     ], WeightingFunction → Euclidean
   ];
  mst = ChangeVertices[
    MinimumSpanningTree[g], (* calculates minimum spanning tree in the graph g *)
    pts];
  MyShowGraph[mst, PlotRange → {{-2, 2}, {-2, 2}}]
 ],
 {{pts, {{1, 0}, {0, 1}, {-1, 0}}}, Locator, LocatorAutoCreate → True},
 SaveDefinitions → True]
```



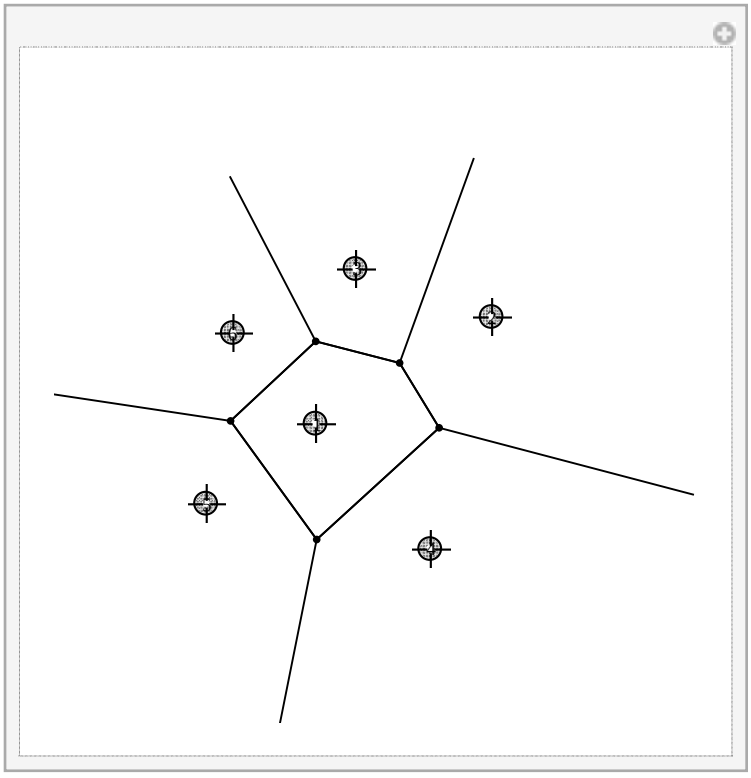
Voronoi diagram

Given M = $\{m_1, \ldots, m_k\} \subset \mathbb{R}^n$, represent $\mathbb{R}^n$ as the union of *cells* V_i, $i = 1, \ldots, k$, where V_i consists of all points $x \in \mathbb{R}^n$ which are not far from m_i than from all other m_j. The point m_i is called the *center* of the cell V_i. The family $\{V_i\}$ is called *Voronoi diagram.*

Algorithms

Mathematica Implementation

```
Needs["ComputationalGeometry`"]

Manipulate[
 DiagramPlot[pts, PlotRange → {{-2, 2}, {-2, 2}}],
 {{pts, {{1, 0}, {0, 1}, {-1, 0}}}, Locator, LocatorAutoCreate → True},
 SaveDefinitions → True, Initialization :→ AbortProtect[Needs["ComputationalGeometry`"]]]
```



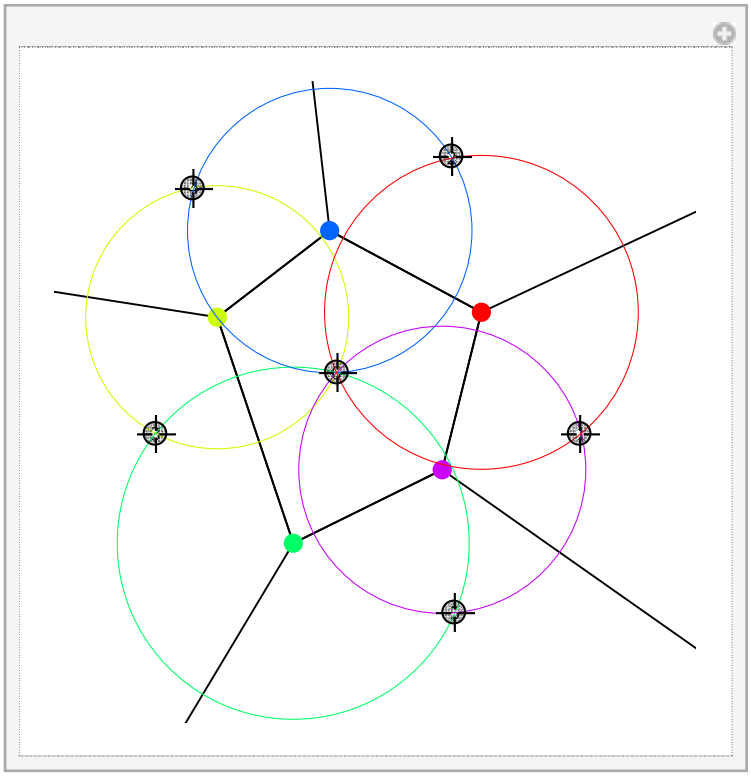
Observations

We consider the planar case, i.e., when M $\subset \mathbb{R}^2$. The union of the boundaries of Voronoi diagram cells V_i is a union of segments and rays which are called the *edges* of the diagram. The points where the edges meet each other are called the *vertices* of the diagram.

1) Let p be a vertex of Voronoi diagram, and V_i, V_j, V_k, ... be the cells containig p. Then the centers m_i, m_j, m_k, ... of these cells lie on the same circle centered at p. We denote such a circle by C(p).

Proof. These points lie at the same distance from p by definition of Voronoi diagram.

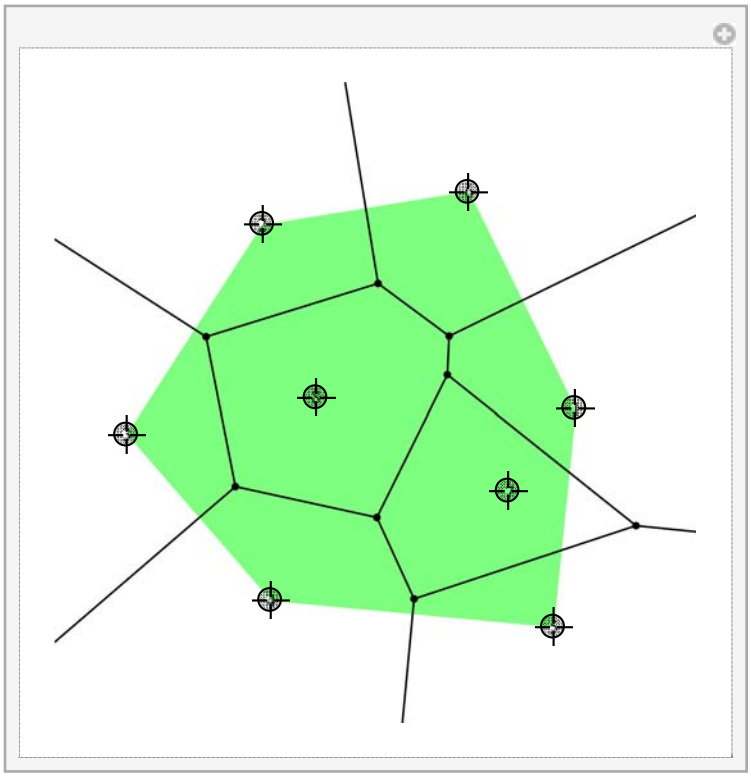
Corollary. If no four points of the original set M are cocircular, then every vertex of the Voronoi diagram is the common intersection of exactly three edges of the diagram.



2) For every vertex p of the Voronoi diagram of M, the disk bounded by the circle C(p) contains no other points of M.
Proof. Other points lie farther from p by definition of Voronoi diagram.

3) Cell V_i is unbounded if and only if m_i is a point on the boundary of the convex hull of the set M.
Exercise. Prove this observation.

```
Manipulate[
 Show[{Graphics[{Green, Opacity[0.5], Polygon[pts[[ConvexHull[pts]]]]}], DiagramPlot[pts]},
  PlotRange → {{-2, 2}, {-2, 2}}],
 {{pts, {{1, 0}, {0, 1}, {-1, 0}}}, Locator, LocatorAutoCreate → True},
 SaveDefinitions → True, Initialization :> AbortProtect[Needs["ComputationalGeometry`"]]]
```



- **Applications**

1) In archaeology, Voronoi polygons are used to map the spread of the use of tools in ancient cultures and for studying the influence of rival centers of commerce [Hodder - Orton (1976)].
2) In ecology, the survival of an organism depends on the number of neighbors it must compete with for food and light, and the Voronoi diagram of forest species and territorial animals is used to investigate the effect of overcrowding [Pielou (1977)].
3) The structure of a molecule is determined by the combined influence of electrical and short-range forces, which have been probed by constructing elaborate Voronoi diagrams.
4) Improves the algorithm for minimum spanning tree construction (see below).

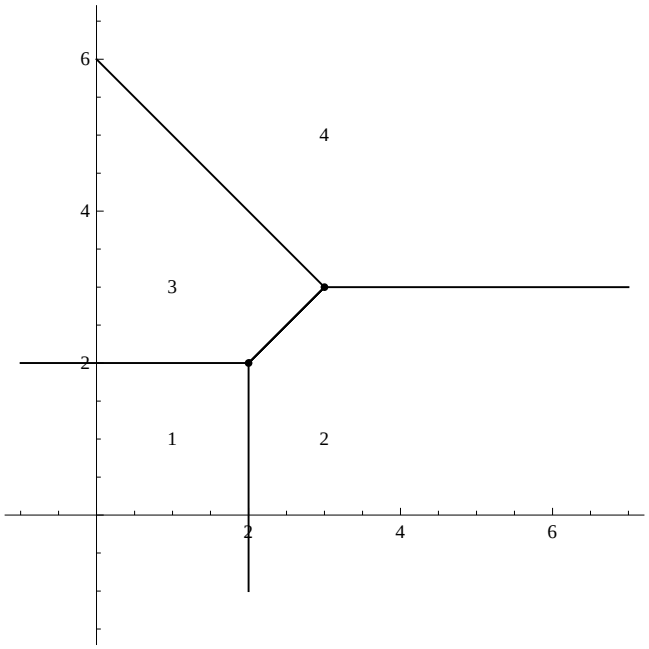
- **Description of *Mathematica* Implementation**

```
Manipulate[
 DiagramPlot[pts, PlotRange → {{-2, 2}, {-2, 2}}],
 (* plots the Voronoi diagram of the points pts = {{x1, y1}, {x2, y2}, ...} *)
 {{pts, {{1, 0}, {0, 1}, {-1, 0}}}, Locator, LocatorAutoCreate → True},
 SaveDefinitions → True, Initialization :> AbortProtect[Needs["ComputationalGeometry`"]]]
```

- **Explanation of *Mathematica* objects used in other applications:**

1) VoronoiDiagram[{{x_1, y_1}, {x_2, y_2}, ...}] yields the planar Voronoi diagram of the points {{x_1, y_1}, {x_2, y_2}, ...}.

```
DiagramPlot[{{1, 1}, {3, 1}, {1, 3}, {3, 5}}, Axes → True]
```




```
VoronoiDiagram[{{1, 1}, {3, 1}, {1, 3}, {3, 5}}]
```

```
{{{2, 2}, {3, 3}, Ray[{2, 2}, {2, -1}], Ray[{2, 2}, {-1, 2}], Ray[{3, 3}, {7, 3}],
  Ray[{3, 3}, {0, 6}]}, {{1, {1, 4, 3}}, {2, {2, 1, 3, 5}}, {3, {1, 2, 6, 4}}, {4, {2, 5, 6}}}}
```

```
VoronoiDiagram[{{1, 1}, {3, 1}, {1, 3}, {3, 5}}][[1]]
(* vertices and rays (infinite edges) of Voronoi diagram *)
```

```
{{2, 2}, {3, 3}, Ray[{2, 2}, {2, -1}],
 Ray[{2, 2}, {-1, 2}], Ray[{3, 3}, {7, 3}], Ray[{3, 3}, {0, 6}]}
```

```
VoronoiDiagram[{{1, 1}, {3, 1}, {1, 3}, {3, 5}}][[2]]
(* vertex adjacency list: an element {i,{v1,...}} corresponds to the point mi = {xi,yi},
and the indices v1, v2,
... identify the vertices or rays which form the boundary of the cell vi centered at mi *)
```

```
{{1, {1, 4, 3}}, {2, {2, 1, 3, 5}}, {3, {1, 2, 6, 4}}, {4, {2, 5, 6}}}
```

■ Playground with *Mathematica* functions

Task 1. Find the vertices of Voronoi diagram for points {{1, 1}, {3, 1}, {1, 3}, {3, 5}}.

```
Module[{pts = {{1, 1}, {3, 1}, {1, 3}, {3, 5}}, vor, numVerts = 0},
 vor = VoronoiDiagram[pts];
 While[! (Head[vor[[1, numVerts + 1]]] === Ray), numVerts++];
 Take[vor[[1]], numVerts]
]
```

```
{{2, 2}, {3, 3}}
```

Task 2. For each vertex p of the previous Voronoi diagram find the circle C(p).

```
Module[{pts = {{1, 1}, {3, 1}, {1, 3}, {3, 5}},
  vor, numVerts = 0, i, j, center, r, circles = {}, dist},
 dist[x_, y_] := Sqrt[(x - y).(x - y)] // N;
 vor = VoronoiDiagram[pts];
 While[! (Head[vor[[1, numVerts + 1]]] === Ray), numVerts++];
 (* calculates the number of vertices in the diagram; see above *)
 For[i = 1, i ≤ numVerts, i++,
  center = vor[[1, i]];
  (* the i-th vertex of Voronoi diagram is the center of the i-th circle *)
  j = 1; (* we look for the point m_j such that V_j contains the vertex vor[[1,i]] *)
  While[! MemberQ[vor[[2, j, 2]], i], j++];
  r = dist[center, pts[[vor[[2, j, 1]]]]]; (* calculate the radius of the i-th circle *)
  circles = circles ~Join~ {Circle[center, r]};
 ];
 circles
]
```

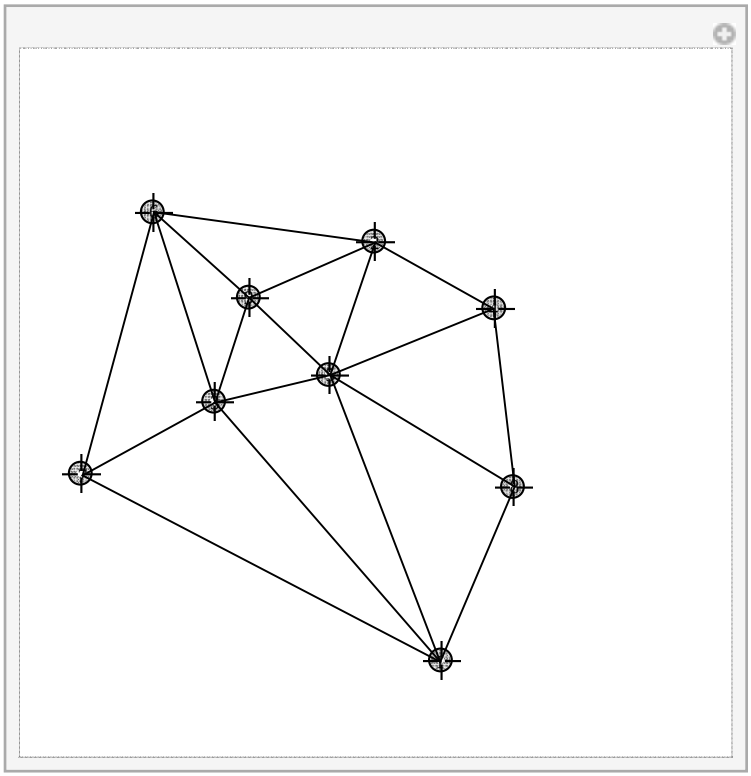
```
{Circle[{2, 2}, 1.41421], Circle[{3, 3}, 2.]}
```

Delaunay triangulation

Given M = $\{m_1, \ldots, m_k\} \subset \mathbb{R}^2$, construct the gual graph G = (M, E) to Voronoi diagram $\{V_i\}$: the pair $\{m_i, m_j\}$ belongs to E iff V_i and V_j intersect each other by more that a point. For M such that any three of its points does not lie on the same straight line or circle, G is a triangulation of the convex hull of M, i.e., drawing edges as correspodning straight segments, G can be considered as a union of triangles each pair of which, if intersected, intersect each other either by a vertex, or by a common edge. The obtaied collection of triangles is called *Delaunay triangulation.* For $M \subset \mathbb{R}^n$ the triangulation consists of simplices.

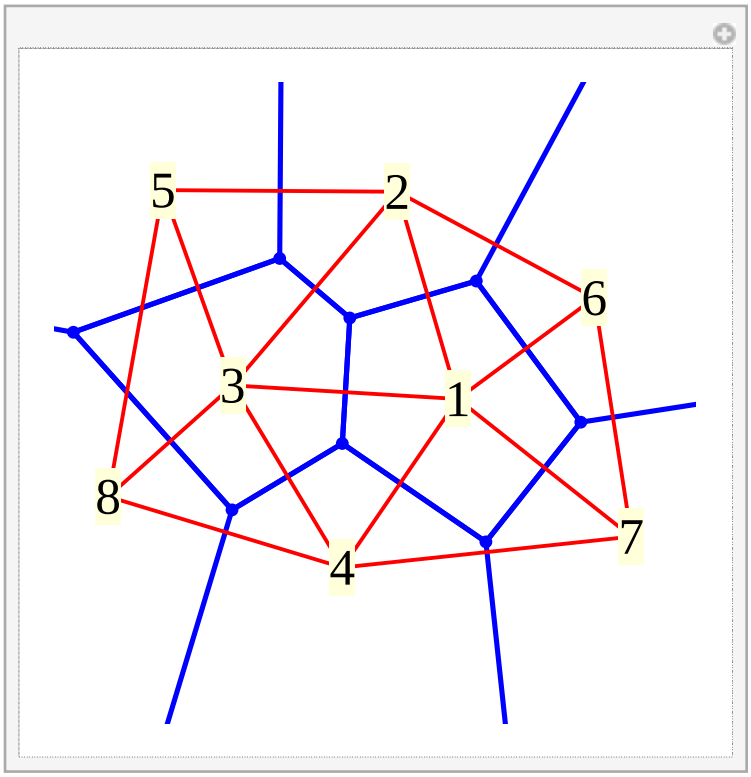
- **Algorithms: Flip algorithms, Incremental, Divide and conquer, Sweepline, Sweephull**
- ***Mathematica* Implementation**

```
Manipulate[
 PlanarGraphPlot[pts, PlotRange → {{-2, 2}, {-2, 2}}],
 {{pts, {{1, 0}, {0, 1}, {-1, 0}}}, Locator, LocatorAutoCreate → True},
 SaveDefinitions → True, Initialization :→ AbortProtect[Needs["ComputationalGeometry`"]]]
```



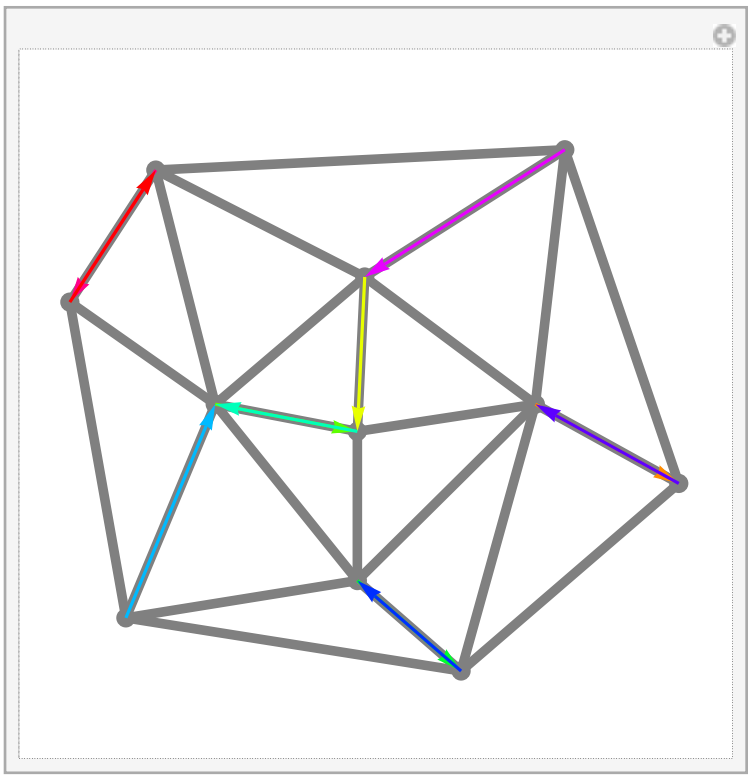
- **The both Voronoi diagram and Delaunay triangulation**

```
Manipulate[Module[{pgp, diag},
  pgp = PlanarGraphPlot[pts];
  diag = DiagramPlot[pts];
  Graphics[{
     Blue, PointSize[0.02], Thickness[0.008], Rest[diag[[1, 2]]], Rest[diag[[1, 3]]],
     Red, Thickness[0.006], Rest[pgp[[1, 2]]]}~Join~
    {Style[#, Black, Large, Background → LightYellow] & /@ pgp[[1, 1]]},
   PlotRange → {{-2, 2}, {-2, 2}}]
 ],
 {{pts, {{1, 0}, {0, 1}, {-1, 0}}}, Locator, LocatorAutoCreate → True, Appearance → None},
 SaveDefinitions → True, Initialization :→ AbortProtect[Needs["ComputationalGeometry`"]]]
```

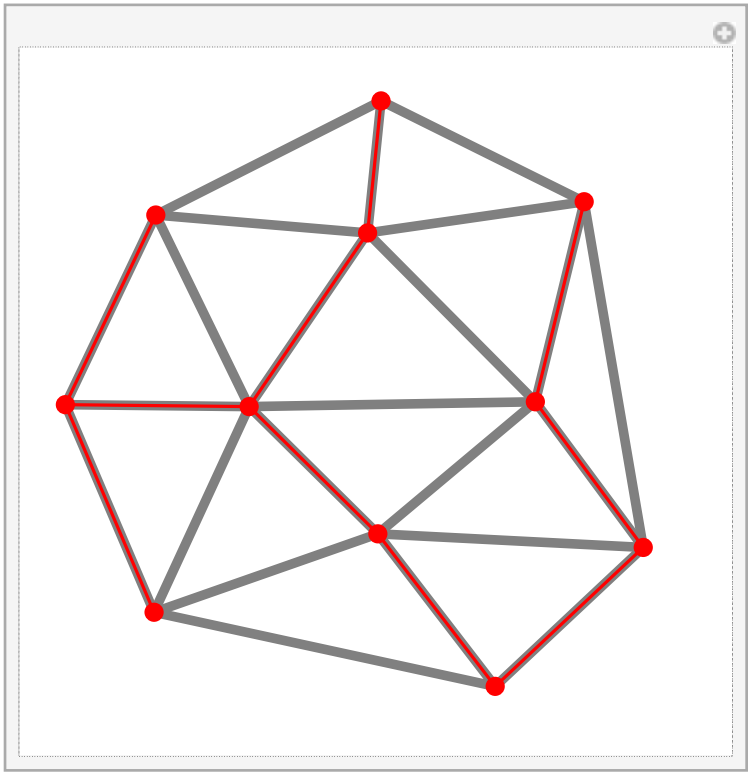


- **Observations**

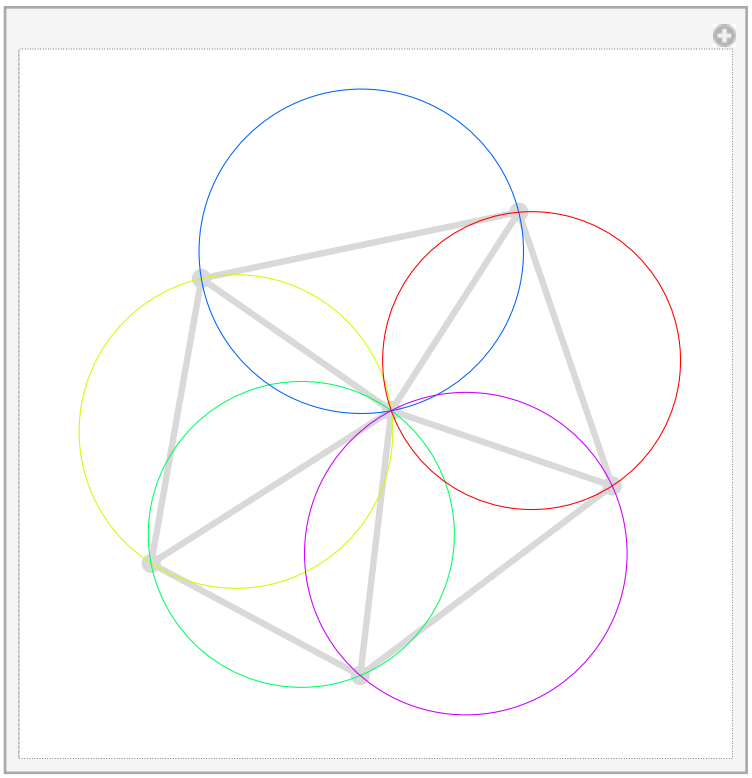
1) The nearest points graph belongs to Delaunay triangulation.



2) Minimum spanning tree belongs to Delaunay triangulation.



3) The disk bounded by a circle around any triangle of Delaunay triangulation does not contain other points from M.

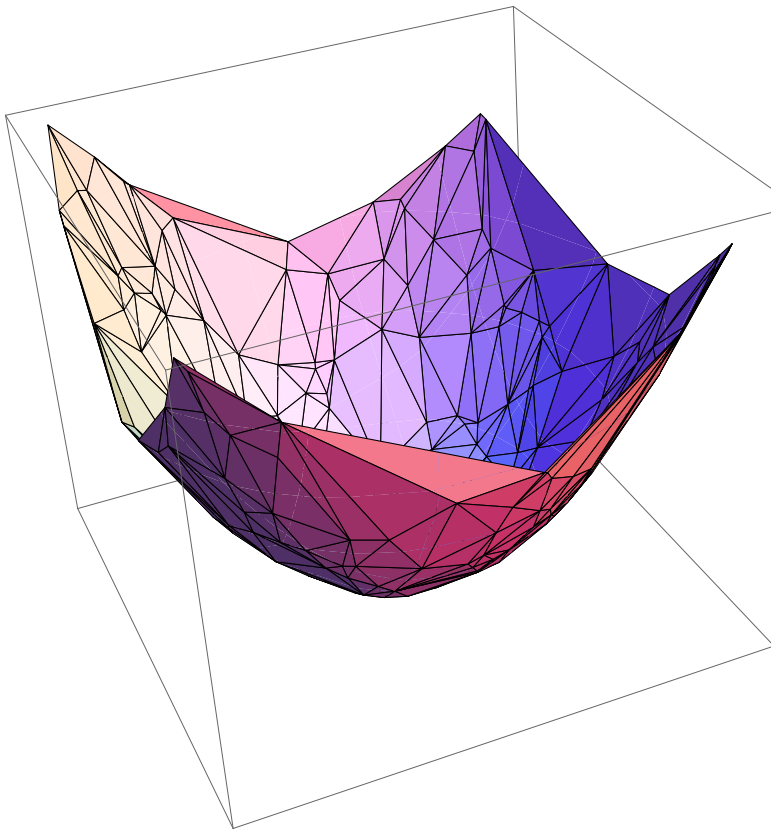


4) Delaunay triangulation maximizes the minimum angle. Compared to any other triangulation of the points, the smallest angle in the Delaunay triangulation is at least as large as the smallest angle in any other. However, the Delaunay triangulation does not necessarily minimize the maximum angle.

- **Applications**

1) Improves the algorithm for minimum spanning tree construction.
2) Improves the algorithm for the nearest points graph construction.
3) For functions aproximations: Delaunay triangulation is the best triangulation in the sense of property 4). This means that Delaunay triangulation avoids narrow triangles.

```
Module[{points, numPoints = 200, values},
 points = Table[{RandomReal[{-1, 1}], RandomReal[{-1, 1}]}, {numPoints}];
 TriangularSurfacePlot[(#1 ~ Join ~ {#1.#1}) & /@ points]
 (* plots the surface according to the Delaunay triangulation
   established by projecting the points onto the xy−plane *)
]
```



4) For modeling terrain or other objects given a set of sample points, the Delaunay triangulation gives a nice set of triangles to use as polygons in the model.

■ **Exercise**

Prove Observations.

■ **Description of *Mathematica* Implementation**

```
Manipulate[
 PlanarGraphPlot[pts, PlotRange → {{-2, 2}, {-2, 2}}],
 (* plots the Delaunay triangulation of the points pts= {{x1,y1},{x2,y3},…} *)
 {{pts, {{1, 0}, {0, 1}, {-1, 0}}}, Locator, LocatorAutoCreate → True},
 SaveDefinitions → True, Initialization :→ AbortProtect[Needs["ComputationalGeometry`"]]]
```