České vysoké učení technické v Praze

Fakulta stavební

155ADKI Algoritmy digitální kartografie a GIS

Konvexní obálky

Bc. Adriana Brezničanová, Bc. Martin Kouba

09. 11. 2020

Úloha č. 2: Konvexní obálky a jejich konstrukce

Vstup: množina $P = \{p_1, ..., p_n\}$, $p_i = [x, y_i]$.

Výstup: $\mathcal{H}(P)$.

Nad množinou P implementujete následující algoritmy pro konstrukci $\mathcal{H}(P)$:

- Jarvis Scan,
- Quick Hull,
- Swep Line.

Vstupní množiny bodů včetně vygenerovaných konvexních obálek vhodně vizualizujte. Pro množiny $n \in < 1000, 1000000 >$ vytvořte grafy ilustrující doby běhu algoritmů pro zvolená n. Měření proveďte pro různé typy vstupních množin (náhodná množina, rastr, body na kružnici) opakovaně (10x) a různá n (nejméně 10 množin) s uvedením rozptylu. Naměřené údaje uspořádejte do přehledných tabulek.

Zamyslete se nad problematikou možných singularit pro různé typy vstupních množin a možnými optimalizacemi. Zhodnoťte dosažené výsledky. Rozhodněte, která z těchto metod je s ohledem na časovou složitost a typ vstupní množiny P nejvhodnější.

Hodnocení:

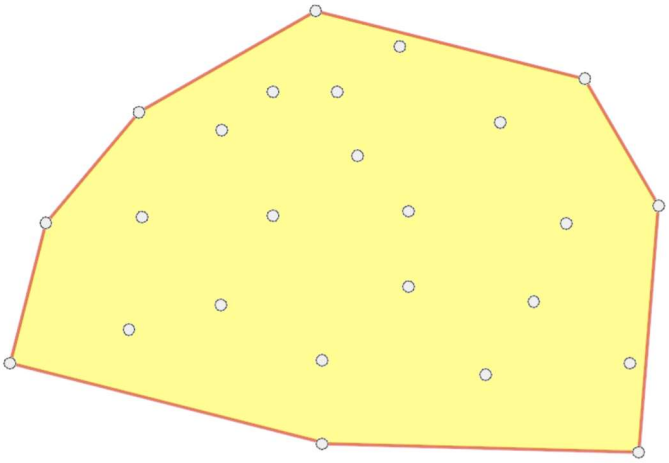
Krok	**Hodnocení**
Konstrukce konvexních obálek metodami Jarvis Scan, Quick Hull, Seep Line.	15b
Konstrukce konvexní obálky metodou Graham Scan	*+5b*
Konstrukce striktně konvexních obálek pro všechny uvedené algoritmy.	*+5b*
Ošetření singulárního případu u Jarvis Scan: existence kolineárních bodů v datasetu.	*+2b*
Konstrukce Minimum Area Enclosing box některou z metod (hlavní směry budov).	*+5b*
Algoritmus pro automatické generování konvexních/nekonvexních množin bodů různých tvarů (kruh, elipsa, čtverec, star-shaped, popř. další).	*+4b*
Max celkem:	**36b**

Čas zpracování: 3 týdny.

Obsah

1. Popis a rozbor problému

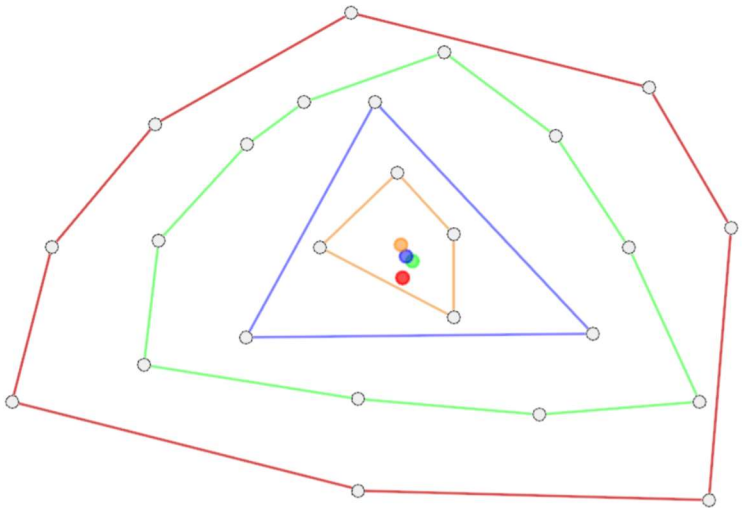
Cílem této úlohy je vytvořit aplikaci pro vytváření konvexních obálek kolem množiny bodů a porovnání dob běhu algoritmů při různě početných množinách. Tyto množiny bodů se budou vkládat ručně pomocí kurzoru nebo se budou náhodně generovat v různých tvarech (kružnice, pravidelná čtvercová síť, náhodné uspořádání). Pro vytvoření konvexních obálek se použijí různé algoritmy. V našem případě byly použity algoritmy Jarvis Scan, Quick Hull a Sweep Line.



Obrázek 1 Konvexní obálka

Definice: Konvexní obálka H množiny konečné množiny S představuje konvexní mnohoúhelník P s nejmenší plochou.

Konvexní obálky se používají například pro plánování pohybu robotů. Dále se využívají v kartografii pro detekci tvaru a natočení budov nebo ve statistice k odhadu centroidu pomocí metody Onion Peeling.



Obrázek 2 Onion Peeling

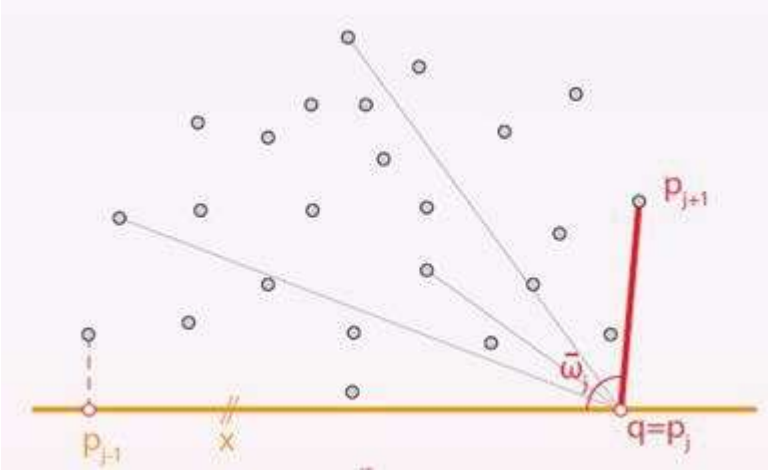
2. Popis algoritmů

2.1 Jarvis Scan

Jako první se hledá pivot *q* - bod s minimální souřadnicí *Y*. Dále se vedou polopřímky ke každému bodu a hledáme maximální úhel (ω_j). Bod s maximálním úhlem ω se přidá do konvexní obálky. Nalezený bod se pak stává pivotem a postup je stejný. Jedná se o opětovné hledání maxima. Výpočet se provádí v cyklu a ten běží, dokud $p_{j+1} \neq q$.
Nevýhoda tohoto algoritmu je, že se nedá použít pro velké data sety.

Výpočet ω_i pomocí dvou směrových vektorů $\vec{u} = |p_{j-1}, p_j|$ a $\vec{v} = |p_j, p_i|$.

$\omega = \text{acos}\left(\frac{\vec{u}\cdot\vec{v}}{\|u\|\cdot\|v\|}\right)$, kde p_{j-1} a p_j jsou body konvexní obálky, bod p_i je bod z množiny bodů.



Obrázek 3 Princip Jarvis Scan

Postup výpočtu algoritmu:
1. Nalezení pivotu *q*, $q = min(y_i)$
2. Přidání *q* do konvexní obálky
3. Inicializace $p_{j-1} \in X$, $p_j = q$, $p_{j+1} = p_{j-1}$
4. Cyklus For $p_{j+1} \neq q$:
   Nalezení $p_{j+1} = argmax_{\forall pi \in P} \angle(p_{j-1}, p_j, p_i)$
   Přidání p_{j+1} do konvexní obálky
   $p_{j-1} = p_j$; $p_j = p_{j+1}$

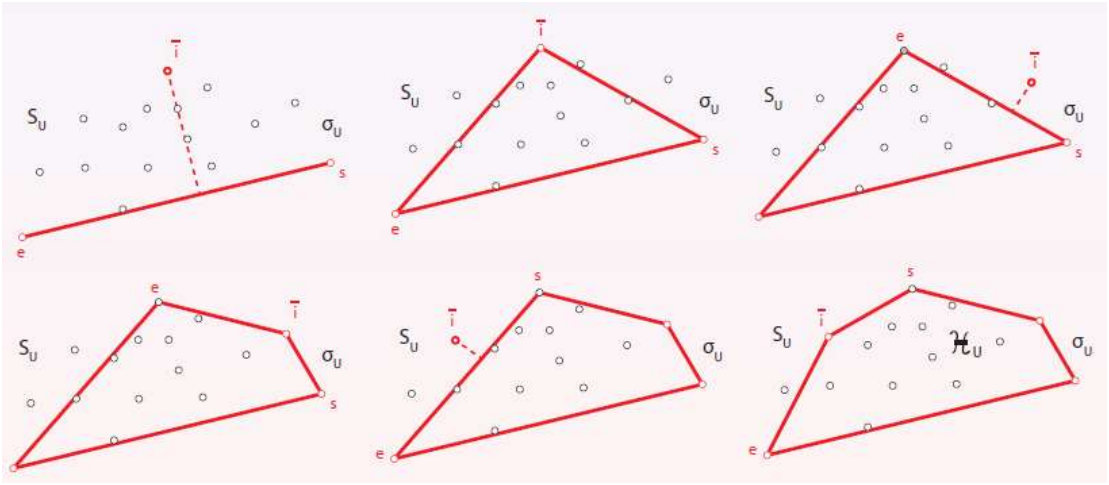
2.2 Quick Hull

Jeden z nejrychlejších algoritmů na tvorbu konvexní obálky, co je jeho výhodou spolu s poměrně lehkou implementací.
Prvním krokem je setřídění množiny bodů dle X. Pak se hledá minimální a maximální hodnota X – extrémní body q_1 a q_3. Spojnice těchto bodů nám rozdělí množinu na dvě části S_U (horní) a S_L (dolní) vzhledem k (q_1, q_3). Každá z těchto částí se řeší zvlášť a na konci se spojí. Při CCW orientaci není nutné provádět spojování.
Algoritmus je rozdělen do dvou procedur – lokální a globální.
Lokální je založena na vyhledávaní nejvzdálenějšího bodu vpravo od přímky a přidání takového bodu do konvexní obálky. Tím vzniknou další spojnice, od kterých se opět hledají nevzdálenější body vpravo od přímek. Toto se provádí rekurzivně a to je důvodem rychlosti algoritmu.

Postup výpočtu algoritmu:
Lokální procedura:
1. Nalezení bodu $\bar{p}$ vpravo od spojnice p_s,p_e s maximální vzdáleností.
2. Pokud $\bar{p} \neq \emptyset$ (nalezen bod vpravo od hrany)
   rekurzivní volání – funkce volá sama sebe, *CH(s, ī, S, K)* // Horní část
   přidání $\bar{p}$ do konvexní obálky
   rekurzivní volání – funkce volá sama sebe, *CH(ī, e, S, K)* // Dolní část

Kde *S* je množina bodů, *s* je index počátečního bodu spojnice, *e* je index koncového bodu spojnice a $\bar{\imath}$ je index nejvzdálenějšího bodu od spojnice.

Globální procedura:
1. Inicializace $K = \emptyset$, $S_U = \emptyset$, $S_L = \emptyset$
2. Setřídění bodů podle *X*
3. Nalezení extrémů $q_1 = min_{\forall pi \in S}(x_i)$, $q_3 = max_{\forall pi \in S}(x_i)$
4. Přidání q_1 a q_3 do S_U a S_L
5. Cyklus For $\forall pi \in S$
   If ($p_i \in \sigma_l(q_1, q_3)$) $p_i \rightarrow S_U$
   else $p_i \rightarrow S_L$
6. Přidání bodu q_3 do konvexní obálky
7. rekurzivní volání – funkce volá sama sebe, *CH(1, 0, S_U, K)* // Horní část
8. Přidání bodu q_1 do konvexní obálky
9. rekurzivní volání – funkce volá sama sebe, *CH(0, 1, S_L, K)* // Dolní část

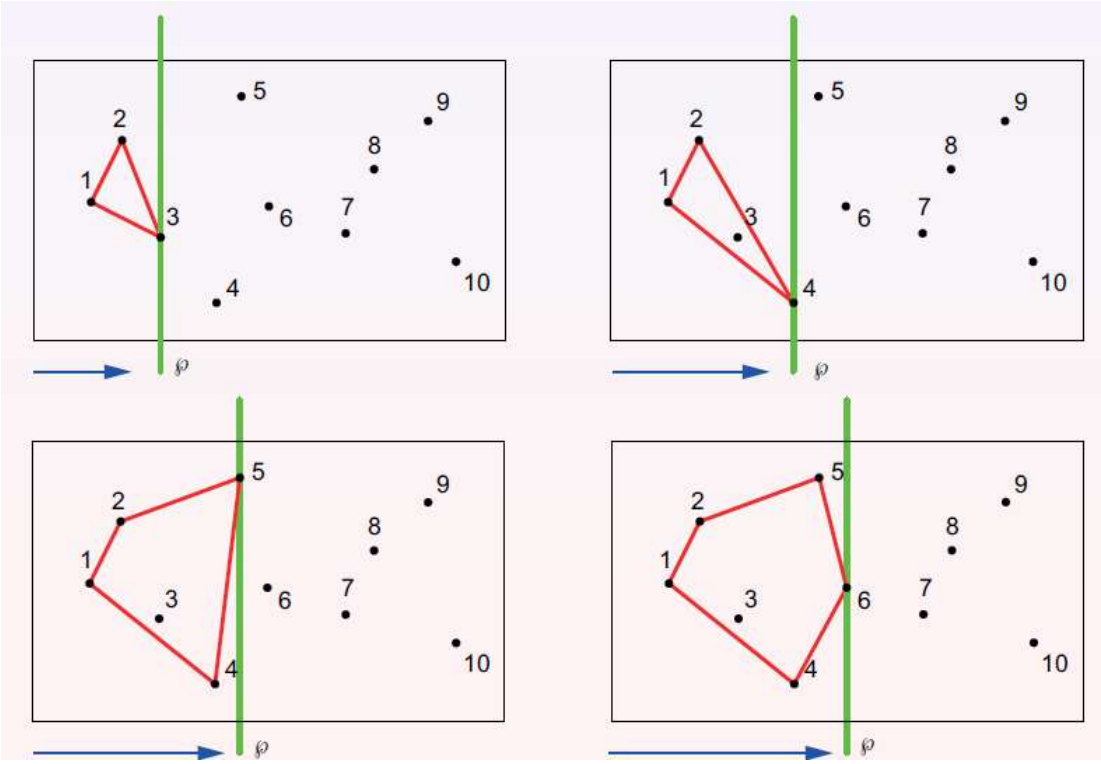


Obrázek 4 Princip Quick Hull

2.3 Sweep Line

Výhodou je rychlost a lze ji převést do vyšší dimenze. Předpokládá se že tvar konvexní obálky se mezi dvěma iteracemi příliš nezmění.
Body se do obálky nepřidávají náhodně, ale v určitém pořadí (dáno zametací přímkou).
Zametací přímkou rozumíme nějakou nadrovinu (my ji chápeme jako přímku), která se nad naší množinou pohybuje zleva doprava a dělí naši množinu bodů na dvě části – na část zpracovanou (vlevo) a část která zpracovaná ještě není (vpravo). Přímka se pohybuje diskrétně – po jednotlivých bodech. Pohybuje se po bodech setříděných podle souřadnice X – předzpracování algoritmu.
Tato metoda nemá ráda, když jsou v množině duplicitní body.
Používá se datový model založen na předchůdcích a následnících – jejich seznamech.

Dále se vytváří iniciální řešení (trojúhelník nebo dvojúhelník) a algoritmus se dělí na dvě iterativní fáze – prvá je vytvoření nekonvexní obálky a druhá převod na konvexní obálku (nekonvexní vrcholy jsou vynechány).



Obrázek 5 Princip Sweep Line

Postup výpočtu algoritmu:

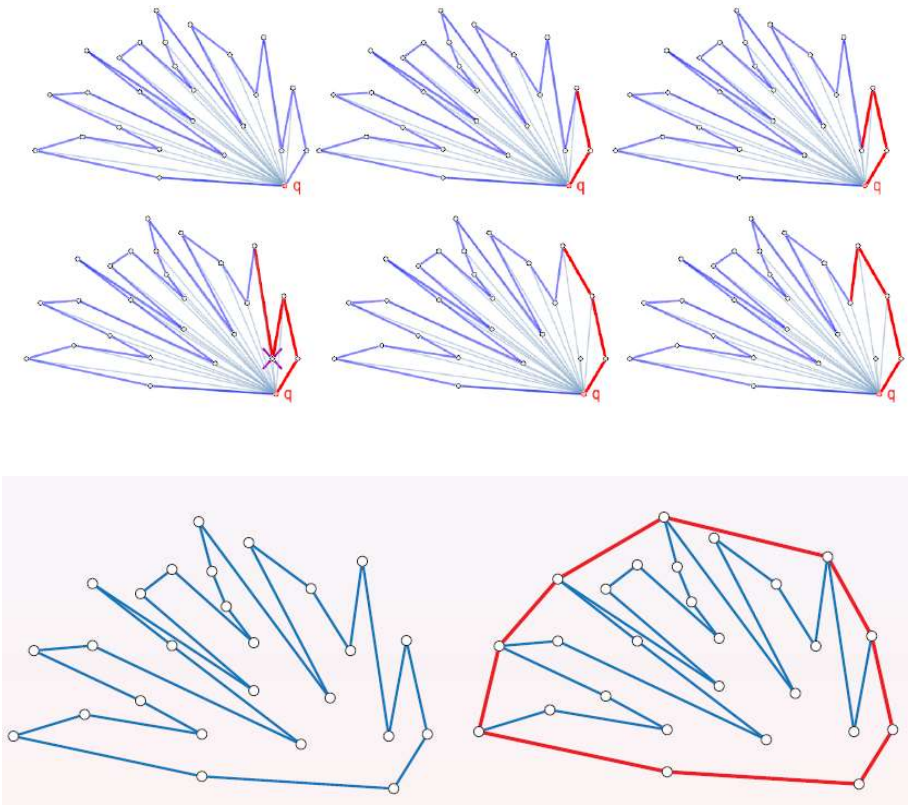
1. Setřídění množiny bodů podle *X*
2. uniquePoints // Odstranění duplicitních bodů
3. Inicializace *n[0] = 1;* // následník
   n[1] = 0;
   p[0] = 1; // předchůdce
   p[1] = 0;
4. For cyklus přes všechny body množiny od *i = 2*
   if $y_i > y_{i-1}$
   p[i] = i-1;
   n[i] = n[i-1];
   else
   p[i] = p[i-1];
   n[i] = i-1;
   Propojení předchůdce a následníka
   p[n[i]] = i;
   n[p[i]] = i;
5. Pokud není horní tečna *n[n[i]]* vpravo od *(i, n[i])*
   p[n[n[i]]] = i;
   n[i] = n[n[i]];
6. 5. Pokud není dolní tečna *p[p[i]]* vlevo od *(i, p[i])*
   n[p[p[i]]] = i;
   p[i] = p[p[i]];

2.4 Graham Scan

Princip tohoto algoritmu je v konstrukci star-shaped polygonu z pivotu *q*. Algoritmus není možné rozšířit do $\mathbb{R}^3$, ale je možné ho použít pro rozsáhlé datasety.
Prvním krokem v tomto algoritmu je nalezení bodu s nejnižší souřadnicí *Y*. Pokud nejnižší souřadnice *Y* existuje ve více než jednom bodě datasetu, měl by být vybrán bod s nejnižší souřadnicí *X*. Takto nalezený bod je pivotem. Dále musí být dataset setříděn ve vzestupné pořadí úhlu, který jednotlivé body tvoří s pivotem a rovnoběžkou s osou *x*. Bod s maximálním úhlem je přidán do konvexní obálky spolu s pivotem. Při přidávání bodů do konvexní obálky musí být splněna podmínka levotočivosti.

Postup výpočtu algoritmu:
- bylo využito zásobníku *S* a jeho funkcí pop() a push().

1. Nalezení pivotu *q*, $q = min(y_i)$, $q \in \mathcal{H}$
2. Setřídění bodů podle úhlu $\omega = \angle(p_j, q, x)$
3. Pokud $\omega_k = \omega_l$, vymaž bod p_k, p_l bližší ke *q*
4. Inicializuj $j = 2$, $S = \emptyset$
5. $S \leftarrow q$, $S \leftarrow p_1$
6. Opakuj pro j < n:
    if p_j vlevo od p_{t-1}, p_t:
        $S \leftarrow p_j$ (vlož p_j ho $\mathcal{H}$)
        $j = j + 1$
    else S.pop() (vyřaď poslední bod z $\mathcal{H}$)



Obrázek 6 Princip Graham Scanu

3. Další součásti aplikace

3.1. Generování kružnice

Ve funkci na generování bodů ve tvaru kružnice bol definován střed – umístěn ve středu canvasu, uhel (φ) určující vykreslování bodů - $2\pi/n$ (n je počet vstupních bodů). Body se následně generují pomocí rovnic pro výpočet souřadnic na kružnici:

$$x = x_0 + r \cdot \cos\varphi$$

$$y = y_0 + r \cdot \sin\varphi$$

3.2. Generování pravidelné čtvercové sítě

Počáteční bod je volen v horním levém rohu. Počet bodů ve sloupcích a řádcích je dán odmocninou z počtu bodů. Rozestup mezi body je dán experimentální hodnou.

3.3. Generování náhodných bodů

Funkce vygeneruje náhodné souřadnice bodů pomocí funkce *rand()* v rozmezí 50 – 549.

3.4. Konstrukce striktně konvexní obálky

Striktně konvexní obálka nesmí obsahovat tři po sobě následující body na jedné přímce a také nesmí obsahovat identické body (nutno ponechat jen jeden unikátní bod). Pro odstranění tohoto případu byla vytvořená funkce, která takovéto body nepřidala do konvexní obálky.
Funkce v prvém kroku prochází body polygonu a pokud nalezne 2 identické body, tak jeden z obálky odstraní. Poté jsou procházeny všechny body a funkcí *getLinePosition* se hledají body, které neleží na přímce (tedy funkce se nerovná -1). Tyto body se přidají do konvexní obálky.

3.5. Třída removeByAngle

Tato funkce byla vytvořená pro použití v algoritmu Graham Scan. Porovnává, zda na spojnici pivot a bod p_i neleží žádný další bod. Funkce eliminuje objekty pokud se dna směry rovnají.

3.6. Třída sortByAngle

Funkce setřídí všechny body v množině. Pokud nalezne uhly, které mají stejný směr, řadí se podle délky.

$$(\sigma_1 < \sigma_2) \,||\, (\sigma_1 == \sigma_2) \,\&\&\, (\mathrm{d}_1 < \mathrm{d}_2)$$

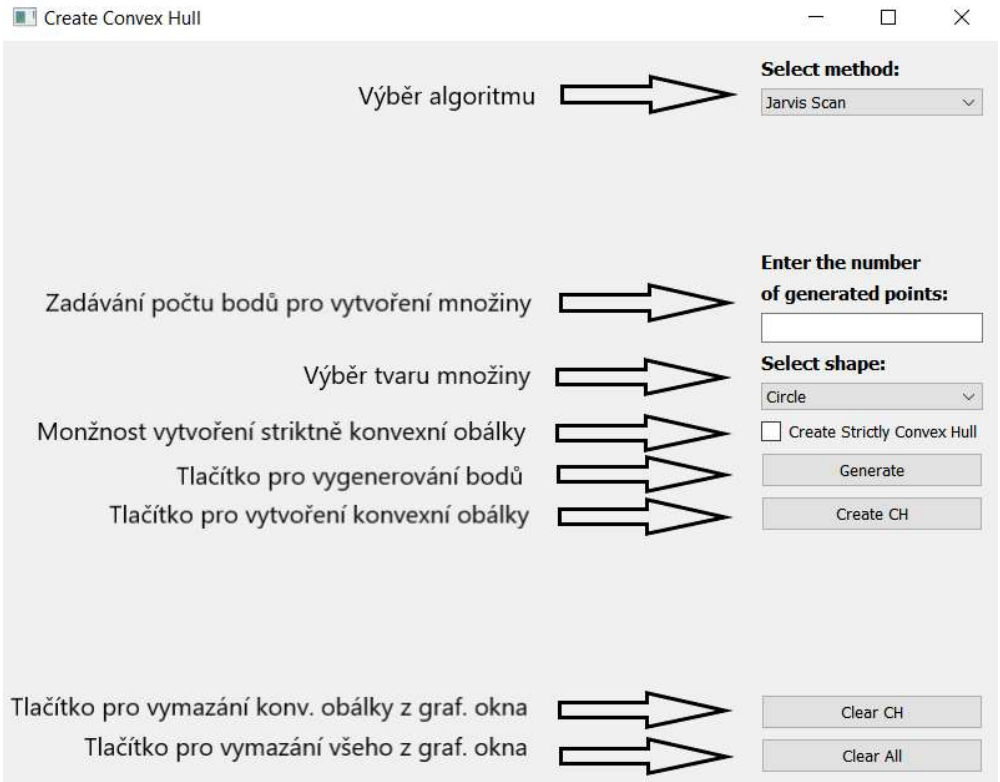
4. Vstupní data

Vstupní data jsou generována podle výběru uživatelem – kružnice, pravidelná mřížka, náhodně, zadáním počtu generovaných bodů *n*. Body je také možné ručně vložit kliknutím myši.
Po kliknutí na tlačítko *Generate* se body vygenerují a zobrazí v Canvasu.

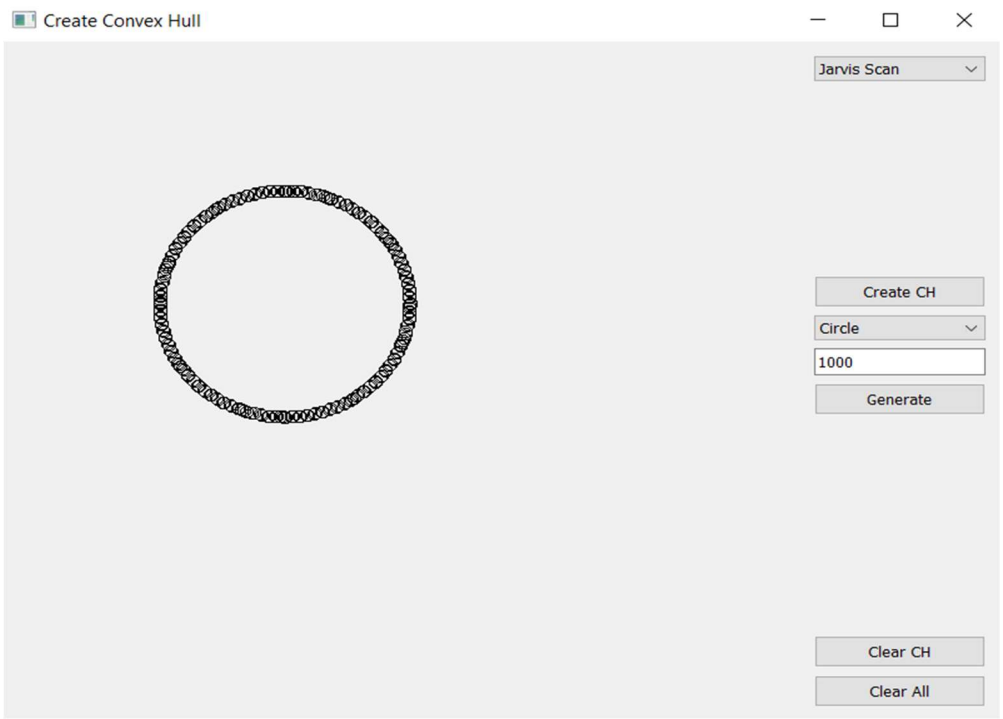
5. Výstupní data

Výstupem rozumíme vykreslení konvexní obálky (zvolenou uživatelem v rolovacím okně) a textový výstup o době trvání algoritmu (v ms) po stlačení tlačítka *Create CH*. Pro tvorbu striktně konvexní obálky slouží „check box“, po jehož označení se vytvoří striktně konvexní obálka. Tento výstup si může uživatel vymazat *Clear CH/ Clear All* a použít uskutečnit další výpočet.

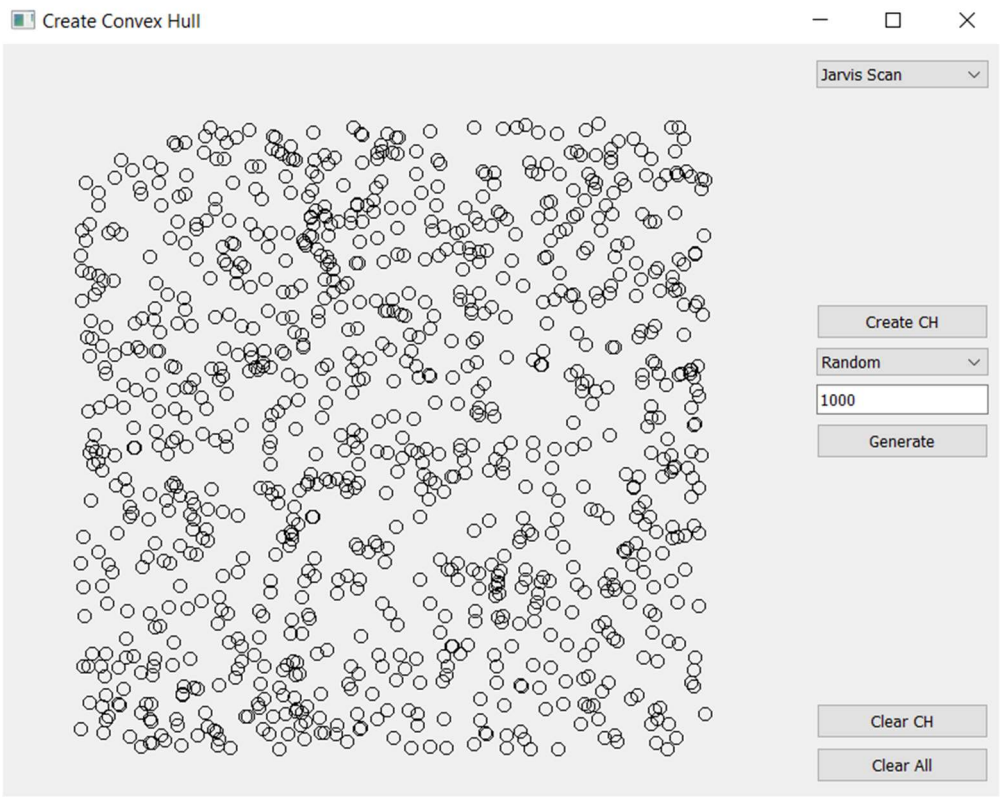
6. Printscreeny z vytvořené aplikace



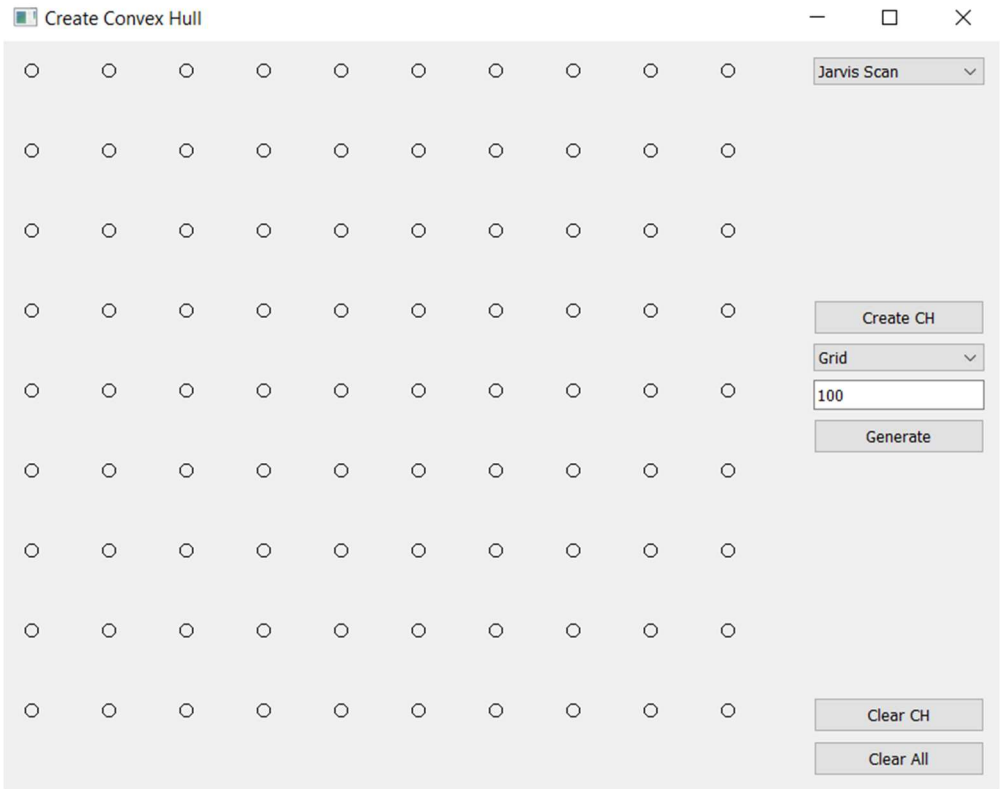
Obrázek 7 Úvodní okno při spuštění aplikace



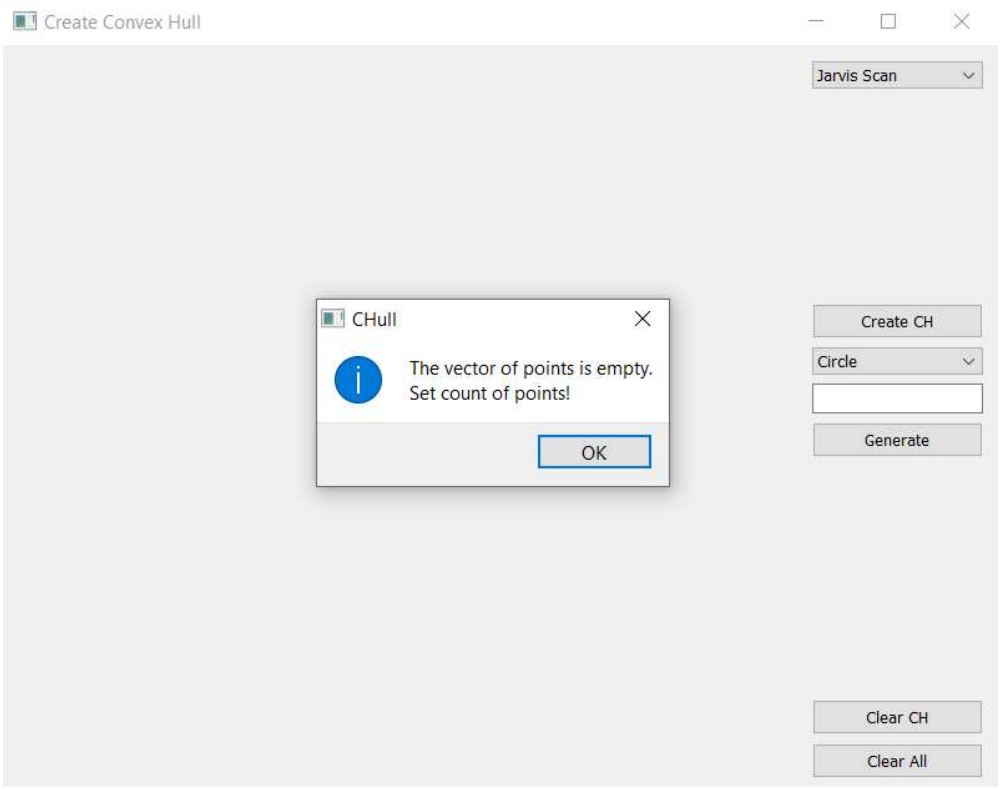
Obrázek 8 Vygenerování kružnice



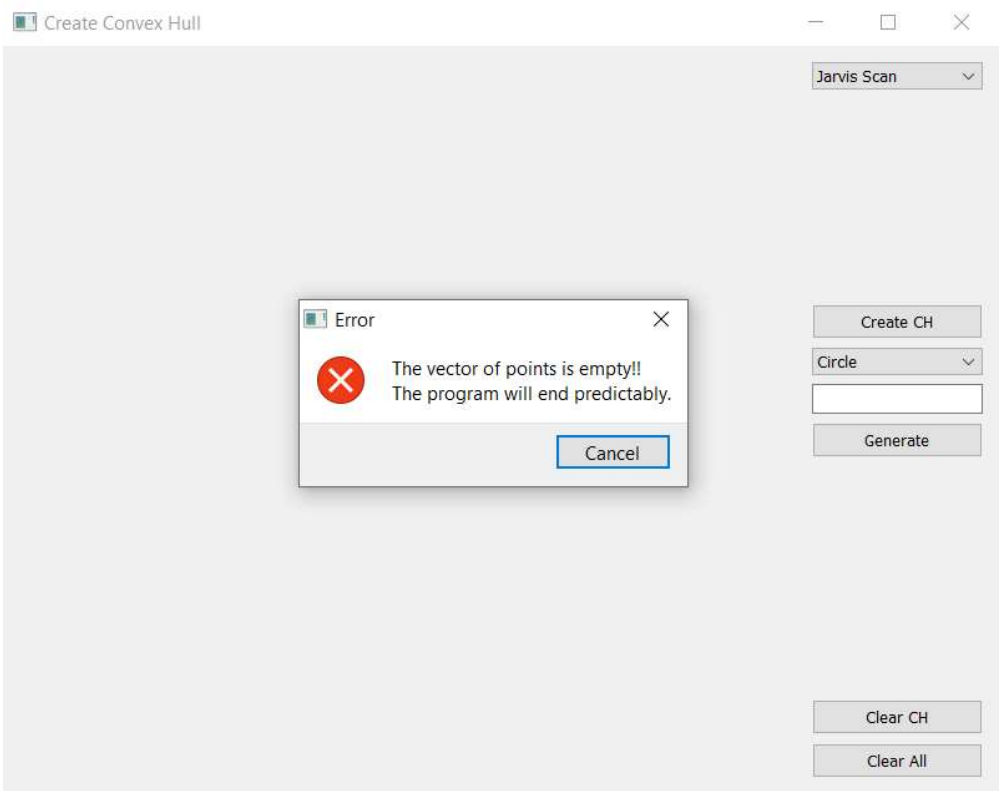
Obrázek 9 Vygenerování náhodných bodů



Obrázek 10 Vygenerování gridu (mřížky bodů)



Obrázek 101 Upozornění při nevyplnění kolonky pro body



Obrázek 92 Error hláška oznamující vypnutí programu po nevytvoření bodů ve snaze vytvořit CH

7. Dokumentace

7.1 Třída algorithms.h

```
class Algorithms
{
 public:
    Algorithms();
// Funkce počítá úhel mezi dvěma vektory
  double getAngle(QPoint &p1, QPoint &p2, QPoint &p3, QPoint &p4);
    // Funkce počítá orientaci bodu vůči přímce a zda se bod nachází v levé
nebo pravé polrovině. Do funkce vstupují souřadnice bodu q a koncové body
přímek.
    static int getPointLinePosition(QPoint &q, QPoint &p1, QPoint &p2);
// Funkce počítá vzdálenost bodu a od přímky danou body p1 a p2
    double getPointLineDist(QPoint &a, QPoint &p1, QPoint &p2);
// Funkce na výpočet konvexní obálky pomocí algoritmu Jarvis Scan
    QPolygon jarvis(std::vector<QPoint> &points);
// Funkce na výpočet konvexní obálky pomocí algoritmu Quick Hull
    QPolygon qhull(std::vector<QPoint> &points);
// Funkce na výpočet konvexní obálky pomocí algoritmu Sweep Line
    QPolygon sweepLine(std::vector<QPoint> &points);
// Funkce na výpočet konvexní obálky pomocí algoritmu Graham Scan
    QPolygon graham(std::vector<QPoint> &points);
// Funkce pro rekurzi pro algoritmus Quick Hull
    void qh(int s, int e, std::vector<QPoint> &points, QPolygon &ch);
// Funkce na tvorbu striktně konvexní obálky - odstranění kolineárních bodů
    static QPolygon strictlyCH(QPolygon &ch);
};
```

7.2 Třída draw.h

```
class Draw : public QWidget
{
    Q_OBJECT
private:
// Deklarace proměnné pro množunu bodů
    std::vector<QPoint> points;     //Input points
// Deklarace proměnné pro konvexní obálku
    QPolygon ch;                    //Convex Hull
public:
// Okno kreslení
    explicit Draw(QWidget *parent = nullptr);
// Funkce pro získání souřadnic bodů z Canvasu (co se stane po kliknutí
myši do Canvasu)
    void mousePressEvent(QMouseEvent *e);
// Funkci na kreslení bodů a konvexní obálky
    void paintEvent(QPaintEvent *e);
// Funkce pro získání souřadnic bodů
    std::vector<QPoint> & getPoints(){return points;}
// Funkce pro získání konvexní obálky
    QPolygon & getCH() {return ch;}
// Funkce pro nastavení konvexní obálky
    void setCH(QPolygon &ch_){ch = ch_;}
// Funkce pro nastavení souřadnic bodů
    void setPoints(std::vector<QPoint> points_){points = points_;}
// Funkce pro generování kružnice
    std::vector<QPoint> circle(int n, int height, int width);
// Funkce pro generování pravidelnej mřížky
    std::vector<QPoint> grid(int n);
// Funkce pro generování náhodných bodů
    std::vector<QPoint> random(int n);
```

```
// Funkce mazáni canvasu
    void clearC();
};
```

7.3 Třída removebyangle.h

```
class removeByAngle{
private:
    QPoint q;

public:
    removeByAngle(QPoint &p){q=p;};
    bool operator() (QPoint &p1, QPoint &p2)
    {
        //Compute direction sigma1, sigma2
        double sigma1 = atan2(p1.y()-q.y(),p1.x()-q.x());
        double sigma2 = atan2(p2.y()-q.y(),p2.x()-q.x());

        //Remove points with same direction
        return sigma1 == sigma2;
    }
};
```

7.4 Třída sortbyangle.h

```
class sortByAngle
{
private:
    QPoint q;
public:
    sortByAngle(QPoint &p){q = p;}
    bool operator()(QPoint &p1, QPoint &p2)
    {
       //Compute directions sigma1, sigma2
        double sigma1 = atan2(p1.y()-q.y(),p1.x()-q.x());
        double sigma2 = atan2(p2.y()-q.y(),p2.x()-q.x());

       //Compute distance d1,d2
        double d1 = (p1.x()-q.x())*(p1.x()-q.x())+(p1.y()-q.y())*(p1.y()-
q.y());
        double d2 = (p2.x()-q.x())*(p2.x()-q.x())+(p2.y()-q.y())*(p2.y()-
q.y());
       //Comparator
        return sigma1 < sigma2 || (sigma1 == sigma2) && (d1 > d2);
    }
};
```

7.5 Třída sortbyx.h

```
class sortByX
{
public:
    sortByX(){};
    //Sorter by x-coordinate
    bool operator () (QPoint &p1, QPoint &p2)
    {
        return p1.x() < p2.x();
    }
};
```

7.6 Třída sortbyy.h

```
class sortByY
{
```

```
public:
    sortByY(){};
    //Sorter by y-coordinate
    bool operator () (QPoint &p1, QPoint &p2)
    {
        return p1.y() < p2.y();
    }
};
```

7.7 Třída widget.h

```
class Widget : public QWidget
{
    Q_OBJECT
public:
    Widget(QWidget *parent = nullptr);
    ~Widget();
private slots:
    void on_pushButton_clicked();
    void on_clear_CH_clicked();
    void on_generate_clicked();
    void on_clear_all_clicked();

private:

        Ui::Widget *ui;
```

7.8 algorithms.cpp

```
double Algorithms::getAngle(QPoint &p1, QPoint &p2, QPoint &p3, QPoint &p4)
{
    //Get vectors u, v
    double ux = p2.x() - p1.x();
    double uy = p2.y() - p1.y();
    double vx = p4.x() - p3.x();
    double vy = p4.y() - p3.y();

    //Norms
    double nu = sqrt(ux * ux + uy * uy);
    double nv = sqrt(vx * vx + vy * vy);

    //Dot product
    double dot = ux * vx + uy * vy;

    return fabs(acos(dot/(nu*nv)));
}


int Algorithms::getPointLinePosition(QPoint &q,QPoint &p1,QPoint &p2)
{
    //Analyze point and line position
    //1 point in the left half plane
    //0 point in the right half plane
    //-1 point on the line
    double ux = p2.x() - p1.x();
    double uy = p2.y() - p1.y();
    double vx = q.x() - p1.x();
    double vy = q.y() - p1.y();
    double t = ux * vy - uy * vx;

    //Point in the left half plane
    if (t>0)
        return 1;
```

```
    if (t<0)
        return 0;
    return -1;
}

double Algorithms::getPointLineDist(QPoint &a,QPoint &p1,QPoint &p2)
{
    //Compute distance of point a from line p(p1, p2)
    double numerator = a.x()* (p1.y() - p2.y()) + p1.x()*(p2.y() - a.y()) +
                       p2.x()*(a.y() - p1.y());
    //Coordinate differences
    double dx = p1.x() - p2.x();
    double dy = p1.y() - p2.y();

    //Point and line distance
    return fabs(numerator)/sqrt(dx*dx + dy*dy);
}

QPolygon Algorithms::jarvis(std::vector<QPoint> &points)
{
    //Create Convex Hull using Jarvis Scan Algorithm
    QPolygon ch;

    //Sort points by y
    std::sort(points.begin(), points.end(), sortByY());

    //Create q
    QPoint q = points[0];

    //Create r
    QPoint r(0,q.y());

    //Initialize pj, pjj
    QPoint pj = q;
    QPoint pjj = r;

    //Add q into Convex Hull
    ch.push_back(q);

    //Find all points of Convex hull
    do {
        //Initialize i_max, om_max
        int i_max = -1;
        double o_max = 0;

        //Find suitable point maximizing angle omega
        for (int i = 0; i < points.size(); i++)
        {
            //Compute omega
            double omega = getAngle(pj, pjj, pj, points[i]);

            //Actualize maximum
            if (omega > o_max)
            {
                o_max = omega;
                i_max = i;
            }
        }

        //Add point to convex hull
        ch.push_back(points[i_max]);
```

```
        //Assign points
        pjj = pj;
        pj = points[i_max];

    } while ((pj!=q));

return ch;
}

QPolygon Algorithms::qhull(std::vector<QPoint> &points)
{
    //Create Convex Hull using QHull Algorithm (Global procedure)
    QPolygon ch;
    std::vector<QPoint> upoints, lpoints;

    //Sort points according to x coordinate
    std::sort(points.begin(), points.end(), sortByX());

    //Create q1, q3
    QPoint q1 = points[0];
    QPoint q3 = points.back();

    //Add q1, q3 to upoints/lpoints
    upoints.push_back(q1);
    upoints.push_back(q3);
    lpoints.push_back(q1);
    lpoints.push_back(q3);

    //Split points to upoints/lpoints
    for (int i = 0; i < points.size(); i++)
    {
        //Add point to upoints
        if (getPointLinePosition(points[i], q1,q3) == 1)
            upoints.push_back(points[i]);

        //Otherwise, add point to lpoints
        else if(getPointLinePosition(points[i], q1,q3) == 0)
            lpoints.push_back(points[i]);
    }

    //Add q3 to CH
    ch.push_back(q3);

    //Recursion for upoints
    qh(1, 0, upoints,ch);

    //Add q1 to CH
    ch.push_back(q1);

    //Recursion for lpoints
    qh(0, 1, lpoints,ch);

    return ch;
}

void Algorithms::qh(int s, int e, std::vector<QPoint> &points, QPolygon
&ch)
{
    //Create Convex Hull using QHull Algorithm (Local procedure)
    int i_max = -1;
```

```
    double d_max = 0;

    //Browse all points
    for (int i = 2; i < points.size(); i++)
    {
        //Point in the right halfplane
        if (getPointLinePosition(points[i], points[s], points[e]) == 0)
        {
            double distance = getPointLineDist(points[i], points[s],
points[e]);

            //Actualize i_max, d_max
            if (distance > d_max)
            {
                d_max=distance;
                i_max=i;
            }
        }
    }

    //Suitable point has been found
    if(i_max!=-1)
    {
        //Process first segment using recursion
        qh(s, i_max, points, ch);

        //Add furthest p to CH
        ch.push_back(points[i_max]);

        //Process second segment using recursion
        qh(i_max, e, points, ch);
    }
}

QPolygon Algorithms::sweepLine(std::vector<QPoint> &points)
{
    //Create Convex Hull using Sweep Line Algortihm
    QPolygon ch;

    //Sort points by X
    std::sort(points.begin(),points.end(),sortByX());

    //Getting unique points from duplicity
    std::vector<QPoint> uniquePoints;
    for (int i =0; i<points.size() - 1; i++)
    {
        if((points[i].x() != points[i+1].x()) || (points[i].y() !=
points[i+1].y()))
        {
            uniquePoints.push_back(points[i]);
        }
    }
    uniquePoints.push_back(points[points.size() - 1]);
    points = uniquePoints;

    //Create lists of predecessors and successors
    int m = points.size();
    std::vector<int> p(m), n(m);

    // Create initial aproximation
    n[0] = 1;
```

```
    n[1] = 0;
    p[0] = 1;
    p[1] = 0;

    // Process all points according to x coordinates
    for (int i = 2; i < m; i++)
    {
        //Point i lies in the upper half plane
        if(points[i].y() >= points[i-1].y()){
            //Link i and its predecessor and successor
            p[i] = i-1;
            n[i] = n[i-1];
        }
        else
        {
            //Link i and its predecessor and successor
            p[i] = p[i-1];
            n[i] = i-1;
        }

        //Remaining links (analogous for both cases)
        p[n[i]] = i;
        n[p[i]] = i;

        //Correct upper tangent
        while (getPointLinePosition(points[n[n[i]]], points[i],
points[n[i]]) == 0)
        {
            //Change predecessor and successor
            p[n[n[i]]] = i;
            n[i] = n[n[i]];
        }

        //Correct lower tangent
        while (getPointLinePosition(points[p[p[i]]], points[i],
points[p[i]]) == 1)
        {
            //Change predecessor and successor
            n[p[p[i]]] = i;
            p[i] = p[p[i]];
        }
    }

    //Conversion of successors to vector
    //Add point with minimum x coordinate
    ch.push_back(points[0]);

    //Get index of its succesor
    int index = n[0];

    //Repeat until first point is found
    while(index != 0)
    {
        //Add to CH
        ch.push_back(points[index]);

        //Get successor
        index = n[index];
    }

    return ch;
```

```
}

QPolygon Algorithms::graham(std::vector<QPoint> &points)
{
    //Create Convex Hull using Graham Algorithm

    std::deque<QPoint> ch;
    QPolygon ch2;

    //Find pivot
    QPoint q = *min_element(points.begin(),points.end(), sortByY());

    //Sort points by their directions
    std::sort(points.begin(), points.end(), sortByAngle(q));

    //Remove duplicate points
    auto it = std::unique(points.begin(),points.end(),removeByAngle(q));

    //Trim vector
    points.resize(it - points.begin());

    //Add 2 points to CH
    ch.push_front(q);
    ch.push_front(points[1]);

    //Process all points
    int j = 2, n = points.size();
    while (j<n)
    {
        //Get point on the top
        QPoint p1 = ch.front();

        //Remove point
        ch.pop_front();

        //Get point on the top
        QPoint p2 = ch.front();

        //Is points[j] in the left halfplane?
        if (getPointLinePosition(points[j],p2,p1)==1)
        {
            //Push point back to Stack
            ch.push_front(p1);

            //Push next point back to Stack
            ch.push_front(points[j]);

            //Increment j
            j++;
        }
    }

    // Adding elements one by one to the vector
    while (!ch.empty())
    {
        ch2.push_back(ch.front());
        ch.pop_front();
    }
    return ch2;
}
```

```
QPolygon Algorithms::strictlyCH(QPolygon &ch)
{
    QPolygon strictly_ch;

    // Check if the first point and the last point are the same
    if (ch[0] == ch.back())
        ch.removeLast();

    int n = ch.size();

    // Process all points
    for (int i = 0; i < n; i++)
    {
        // Check if three points are not collinear
        if (getPointLinePosition(ch[(i+2)%n], ch[i], ch[(i+1)%n]) != -1)
            strictly_ch.push_back(ch[(i+1)%n]);
    }

    return strictly_ch;
}
```

7.9 draw.cpp

```
void Draw::mousePressEvent(QMouseEvent *e)
{
    //Get coordinates
    int x = e->x();
    int y = e->y();

    //Add point to the list
    QPoint p(x,y);
    points.push_back(p);

    //Repaint
    repaint();

}

void Draw::paintEvent(QPaintEvent *e)
{
    //Start draw
    QPainter qp(this);
    qp.begin(this);

    //Draw points
    int r = 5;
    for(int i = 0; i < points.size(); i++)
    {
        qp.drawEllipse(points[i].x() - r,points[i].y() - r, 2 * r, 2 * r);
    }

    //Draw polygon
    qp.drawPolygon(ch);

    //End draw
    qp.end();
}

std::vector<QPoint> Draw::circle(int n,int height, int width){

    std::vector<QPoint> points;
    QPoint p, center;
```

```
    double phi = 2*M_PI/n;

    center.setX(width/2);
    center.setY(height/2);

    for (int i = 0; i < n; i++)
    {
         p.setX(center.x() + (height/2-100) * cos(phi*i));
         p.setY(center.y() + (height/2-100) * sin(phi*i));
         points.push_back(p);
    }
    return points;
}

std::vector<QPoint> Draw::grid(int n){

    QPoint point;

    int count = sqrt(n);

    for (int x = 0; x < count; x++)
    {
        for (int y = 0; y < count; y++)
        {
            point.setX(x*20+10);
            point.setY(y*20+10);
            points.push_back(point);
        }
    }

    return points;
}

std::vector<QPoint> Draw::random(int n){

    std::vector<QPoint> points;
    QPoint p;

    for(int i = 0;i<n;i++){

        double x = rand()%500+50;
        double y = rand()%500+50;
        p.setX(x);
        p.setY(y);
        points.push_back(p);
    }

    return points;
}


void Draw::clearC()
{
    //Clear points
    points.clear();

    //Repaint screen
    repaint();
}
```

7.10 widget.cpp

```
void Widget::on_pushButton_clicked()
{
    //Get points
    std::vector<QPoint> points = ui->Canvas->getPoints();

    if(points.empty()){
        QMessageBox::critical(this,tr("Error"),
                              tr("The vector of points is empty!!\n"
                                 "The program will end
predictably."),QMessageBox::Cancel);
    }

    //Create Convex hull
    QPolygon ch;
    Algorithms alg;

    clock_t start_timer = std::clock();

    //Jarvis Scan
    if (ui->comboBox->currentIndex()==0)
        ch = alg.jarvis(points);
    //QHull
    else if (ui->comboBox->currentIndex()==1)
        ch = alg.qhull(points);
    //Sweep Line
    else if (ui->comboBox->currentIndex()==2)
        ch = alg.sweepLine(points);
    //Graham Scan
    else
        ch = alg.graham(points);

    //Set Convex hull
    ui -> Canvas -> setCH(ch);

    clock_t end_timer = std::clock();

    clock_t time = end_timer-start_timer;

    //Set time
    ui -> label -> setText(QString::number(time) + "ms");

    // Draw strictly convex hull
    if (ui->checkBox->isChecked())
    {
        QPolygon strictly_ch = Algorithms::strictlyCH(ch);
        ui->Canvas->setCH(strictly_ch);
        repaint();
    }
    else
    {
        ui->Canvas->setCH(ch);
        repaint();
    }
}

void Widget::on_clear_CH_clicked()
{
    //Get Convex Hull
    QPolygon &ch = ui->Canvas->getCH();
```

```
    //Clear points
    ch.clear();

    //Repaint screen
    repaint();
}

void Widget::on_generate_clicked()
{
    std::vector<QPoint> points;
    Draw drw;
    int n, height, width;
    if (ui->comboBox_2->currentIndex()==0){
        n = ui->lineEdit->text().toInt();
        height = ui->Canvas->height();
        width = ui->Canvas->width();
        points = drw.circle(n,height,width);
    }
    else if (ui->comboBox_2->currentIndex()==1){
        n = ui->lineEdit->text().toInt();
        points = drw.grid(n);
    }
    else if (ui->comboBox_2->currentIndex()==2){
        n = ui->lineEdit->text().toInt();
        points = drw.random(n);
    }

    if(points.size()==1){
        QMessageBox::information(this,tr(""),
                          tr("The vector of points is empty.\n"
                             "Set count of points!"),QMessageBox::Ok);
    }
    ui->Canvas->setPoints(points);
    ui->Canvas->repaint();
}


void Widget::on_clear_all_clicked()
{

    //Get Convex Hull
    QPolygon &ch = ui->Canvas->getCH();

    //Clear Convex Hull
    ch.clear();

    //Clear points
    ui->Canvas->clearC();

    //Repaint screen
    repaint();
}
```

8. Výsledky

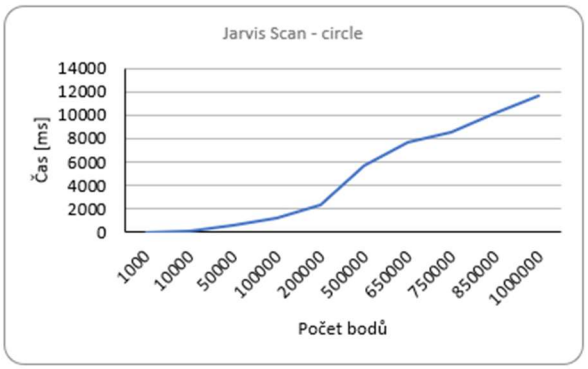
Jak bylo zadáním, tak byla testována časová náročnost algoritmů pro deset zvolených množin bodů desetkrát. Testované byly všechny čtyři algoritmy. Testování probíhalo na množinách ve tvaru kružnice, pravidelné mřížky a náhodně vykreslených bodech. Výsledky jsou zobrazeny v následujících tabulkách a grafech.

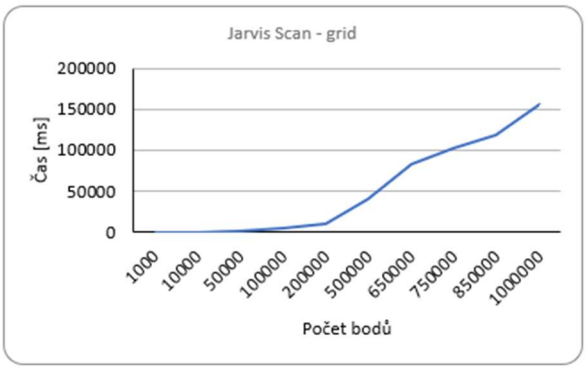
8.1. Jarvis Scan

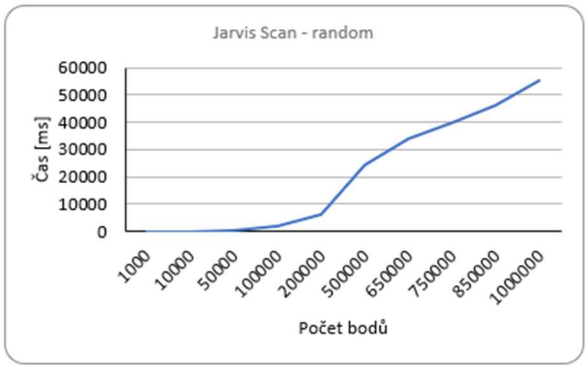
Kružnice										
Počet bodů	**1000**	**10000**	**50000**	**100000**	**200000**	**500000**	**650000**	**750000**	**850000**	**1000000**
Čas [ms]	7	112	565	1139	2329	5711	7602	8498	10181	11604
	8	114	567	1145	2325	5715	7606	8509	10389	11607
	7	113	566	1140	2320	5713	7615	8503	10209	11646
	7	111	566	1144	2327	5726	7621	8505	10188	11640
	7	112	552	1163	2328	5712	7618	8518	10178	11614
	8	112	550	1144	2339	5724	7613	8503	10180	11608
	7	111	566	1149	2324	5725	7604	8510	10182	11599
	7	111	565	1140	2321	5719	7618	8514	10228	11651
	7	111	566	1143	2323	5720	7619	8456	10192	11610
	7	112	568	1144	2320	5737	7608	8499	10196	11607
Průměr	7,2	111,9	563,1	1145,1	2325,6	5720,2	7612,4	8501,5	10212	11618,6
Rozptyl	0,16	0,89	37,49	43,29	29,24	58,56	42,64	266,25	3686,6	333,24

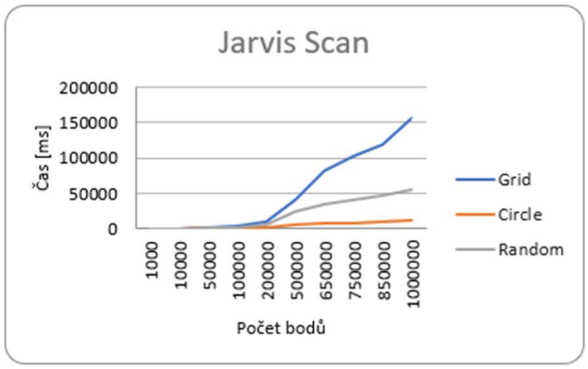
Grid										
Počet bodů	**1000**	**10000**	**50000**	**100000**	**200000**	**500000**	**650000**	**750000**	**850000**	**1000000**
Čas [ms]	5	147	1346	4074	10268	40635	82216	101857	119180	156452
	5	148	1320	4019	10126	40717	82366	102044	119047	156422
	5	146	1329	4076	9968	40506	82360	101862	119183	156877
	6	148	1352	4067	10215	40797	82359	102003	119076	156735
	5	143	1342	4007	10114	40686	82371	101888	119057	156345
	6	151	1357	4056	9958	40546	82274	101914	118962	156506
	6	153	1355	4064	10164	40769	82349	101878	119096	156257
	6	147	1327	4051	10151	39856	82394	102355	118995	156433
	5	150	1352	4089	10113	40774	82348	101815	119181	156593
	5	143	1337	4093	10190	40609	82419	101923	119012	156803
Průměr	5,4	147,6	1341,7	4059,6	10127	40590	82346	101954	119079	156542
Rozptyl	0,24	9,24	151,21	703,24	8774,6	68422	3111,8	22119	5846,1	37590,6

Random										
Počet bodů	**1000**	**10000**	**50000**	**100000**	**200000**	**500000**	**650000**	**750000**	**850000**	**1000000**
Čas [ms]	2	48	560	1924	6147	24550	33400	39421	46077	54790
	2	52	553	1981	6450	24259	34299	40314	46146	57106
	2	38	508	1894	6387	24374	33821	40061	45756	55879
	2	41	632	2165	6619	24396	34180	38949	46032	55377
	3	38	527	2135	6443	24706	33003	39338	46727	55561
	3	37	657	1858	6304	24588	33696	40970	45959	55268
	2	44	525	1847	6599	24055	34748	40037	46325	54927
	3	41	569	1944	6243	24093	34442	39999	46215	55608
	2	38	555	2290	6481	24717	34474	39849	46323	55638
	3	44	534	2059	6432	24990	34025	39306	46453	54788
Průměr	2,4	42,1	562	2009,7	6410,5	24473	34009	39824	46201	55494,2
Rozptyl	0,24	21,89	2040,2	19617	19644	78092	257966	311876	67419	416530









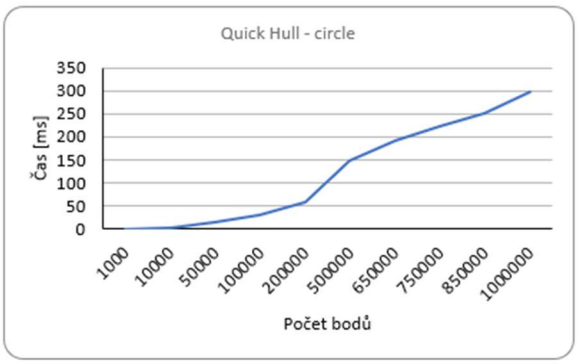
Zde vidět, jak se projevila složitost $O(n^2)$. Čas při milionu bodech oproti 1000 je při kružnici větší přibližně 1600 násobně, při gridu až 29000 násobně a při náhodných bodech 23000 násobně. Z grafů vidět, že od počtu 200000 bodů čas rychle stoupá. Sice se jako nejlepší možnost jeví kružnice, i tak je tento algoritmus nevhodný pro velké množiny.

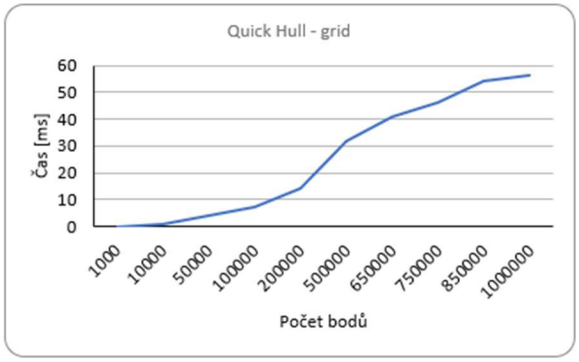
8.2. Quick Hull

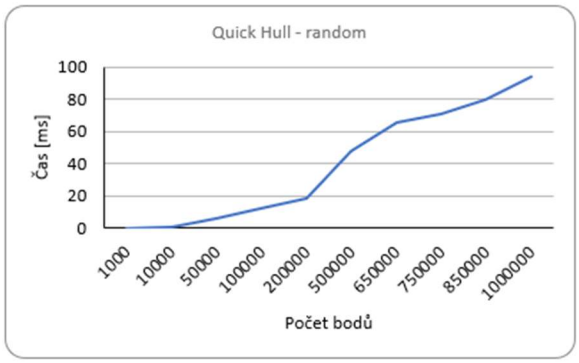
Kružnice										
Počet bodů	**1000**	**10000**	**50000**	**100000**	**200000**	**500000**	**650000**	**750000**	**850000**	**1000000**
Čas [ms]	0	3	15	28	59	146	191	220	250	292
	0	3	15	29	59	147	185	222	253	296
	0	3	15	30	59	148	189	223	250	298
	0	3	15	30	60	150	193	225	251	301
	0	3	15	30	59	153	190	225	250	297
	0	4	14	30	57	148	192	221	256	302
	1	3	15	31	59	149	191	223	251	301
	1	3	15	29	57	150	193	225	251	301
	1	3	15	32	59	147	191	221	248	298
	1	3	15	30	60	148	190	220	250	303
Průměr	0,4	3,1	14,9	29,9	58,8	148,6	190,5	222,5	251	298,9
Rozptyl	0,24	0,09	0,09	1,09	0,96	3,64	4,85	3,65	4,2	10,09

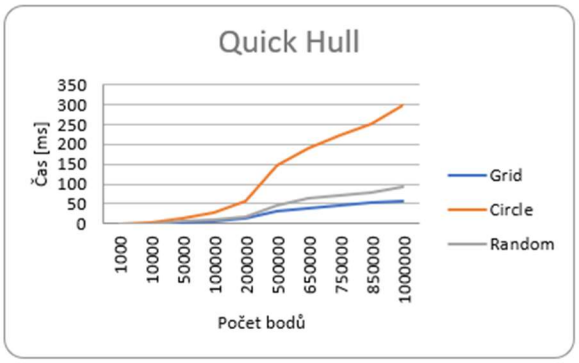
Grid										
Počet bodů	**1000**	**10000**	**50000**	**100000**	**200000**	**500000**	**650000**	**750000**	**850000**	**1000000**
Čas [ms]	0	1	4	7	15	31	44	49	55	59
	0	1	4	6	14	31	40	49	52	61
	0	1	4	7	14	32	40	47	54	56
	0	1	4	7	14	34	43	44	54	54
	0	1	4	8	14	32	42	46	56	56
	0	0	4	7	13	31	41	43	56	55
	0	1	4	8	15	34	41	49	52	56
	0	1	4	7	14	33	38	44	54	56
	0	1	3	7	15	31	40	46	53	54
	0	1	4	8	14	31	38	47	54	54
Průměr	0	0,9	3,9	7,2	14,2	32	40,7	46,4	54	56,1
Rozptyl	0	0,09	0,09	0,36	0,36	1,4	3,41	4,44	1,8	4,69

Random										
Počet bodů	**1000**	**10000**	**50000**	**100000**	**200000**	**500000**	**650000**	**750000**	**850000**	**1000000**
Čas [ms]	0	1	6	12	18	48	67	70	80	94
	0	1	5	12	19	48	66	71	81	86
	1	1	5	12	19	49	64	71	78	96
	0	1	6	11	19	48	65	71	79	93
	0	1	6	12	18	48	65	72	80	94
	0	1	5	12	18	47	65	70	80	96
	0	0	5	12	18	47	65	70	80	92
	0	1	5	12	18	47	64	71	79	100
	0	1	6	12	18	48	66	71	79	96
	0	1	6	12	19	48	66	70	80	94
Průměr	0,1	0,9	5,5	11,9	18,4	47,8	65,3	70,7	79,6	94,1
Rozptyl	0,09	0,09	0,25	0,09	0,24	0,36	0,81	0,41	0,64	11,69









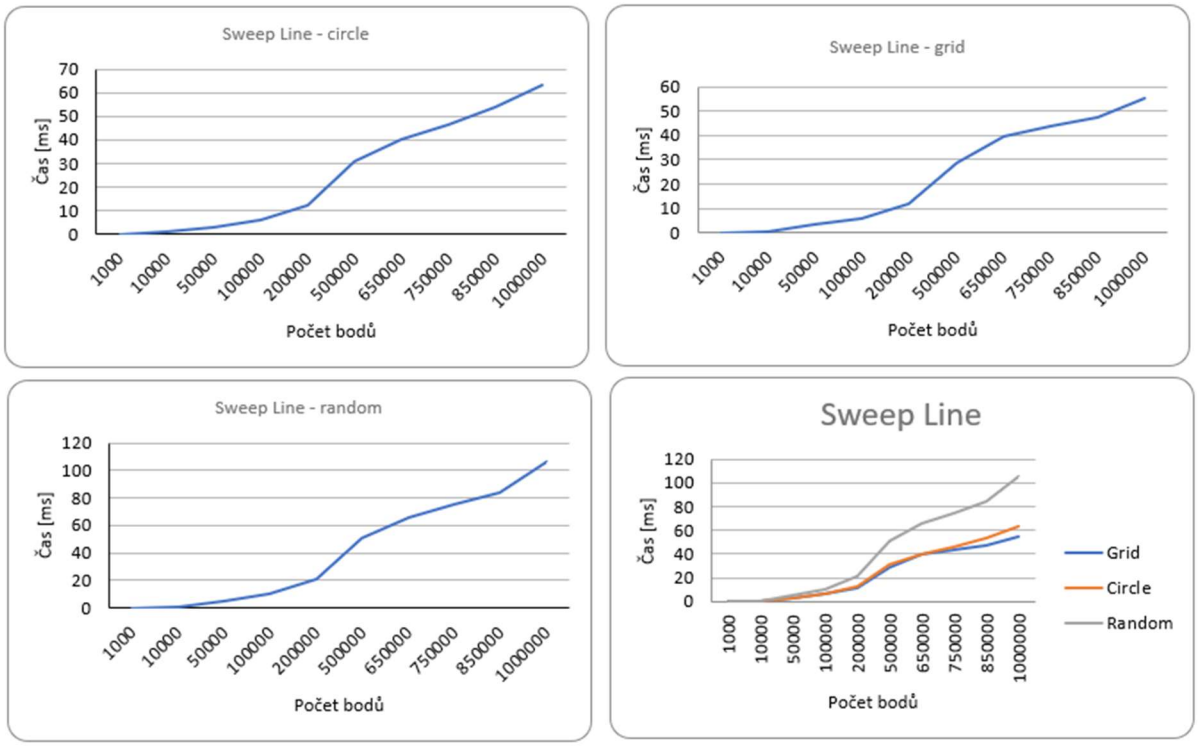
Už na první pohled na tabulky je zřejmé, že složitost algoritmu Quick Hull $O(n \cdot log\ n)$ je vhodná aj pro rozsáhlé datasety. Čas při milionu bodech oproti 1000 je při kružnici větší přibližně 750 násobně, při gridu 60 násobně a při náhodných bodech 950 násobně. Z grafů vidět, že od počtu 200000 bodů čas stoupá, ale je pořád nízký. Jako nejlepší možnost je při Quick Hull algoritmu jeví grid.

8.3. Sweep Line

Kružnice										
Počet bodů	**1000**	**10000**	**50000**	**100000**	**200000**	**500000**	**650000**	**750000**	**850000**	**1000000**
Čas [ms]	0	1	3	6	12	30	40	46	52	63
	0	1	3	6	12	30	41	46	54	63
	0	1	3	6	13	31	41	46	54	62
	0	1	3	7	12	31	40	46	53	63
	0	1	3	6	13	31	40	47	54	63
	0	1	3	6	12	31	39	47	54	63
	0	1	2	6	12	31	39	47	53	65
	0	1	3	6	12	31	41	47	54	63
	0	1	3	6	12	31	40	47	55	64
	0	0	3	6	13	31	41	47	53	62
Průměr	0	0,9	2,9	6,1	12,3	30,8	40,2	46,6	53,6	63,1
Rozptyl	0	0,09	0,09	0,09	0,21	0,16	0,56	0,24	0,64	0,69

Grid										
Počet bodů	**1000**	**10000**	**50000**	**100000**	**200000**	**500000**	**650000**	**750000**	**850000**	**1000000**
Čas [ms]	0	0	3	6	12	31	37	44	46	52
	0	1	3	6	13	26	38	41	42	55
	0	0	3	6	12	28	40	41	47	57
	0	0	3	7	13	27	40	46	45	53
	0	1	4	5	12	30	40	45	49	53
	0	1	3	6	12	29	42	41	46	60
	0	1	4	7	12	30	40	45	49	54
	0	1	3	6	12	26	39	44	49	55
	0	1	3	6	10	30	37	46	52	57
	0	0	3	6	12	29	40	42	49	56
Průměr	0	0,6	3,2	6,1	12	28,6	39,3	43,5	47,4	55,2
Rozptyl	0	0,24	0,16	0,29	0,6	2,84	2,21	3,85	7,04	5,16

Random										
Počet bodů	**1000**	**10000**	**50000**	**100000**	**200000**	**500000**	**650000**	**750000**	**850000**	**1000000**
Čas [ms]	0	1	5	10	21	49	65	74	84	105
	0	1	5	9	21	50	66	75	83	105
	0	1	5	10	21	51	66	74	84	106
	0	1	5	10	21	51	66	75	82	106
	0	1	5	10	21	51	65	74	85	107
	0	1	5	11	22	50	65	75	84	106
	0	1	5	11	21	51	65	75	85	104
	1	0	5	10	21	52	64	76	84	105
	0	1	5	10	21	50	65	76	83	107
	0	1	5	10	21	51	65	75	85	107
Průměr	0,1	0,9	5	10,1	21,1	50,6	65,2	74,9	83,9	105,8
Rozptyl	0,09	0,09	0	0,29	0,09	0,64	0,36	0,49	0,89	0,96



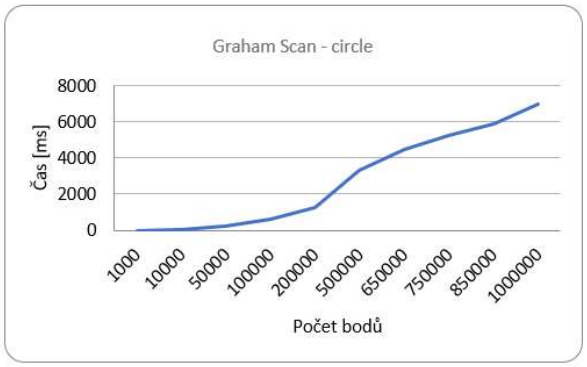
Složitost $O(n \cdot log\ n)$ při algoritmu Sweep line se taktéž ukázala jako výhodná aj pro rozsáhlé datasety. Čas při milionu bodech oproti 1000 je při kružnici a při gridu přibližně stejný, při náhodných bodech je o něco vyšší. Z grafů vidět, že od počtu 200000 bodů čas stoupá, ale je pořád nízký. Jako nejlepší možnost je při Quick Hull algoritmu jeví grid podobně jako při Quick Hull algoritme.

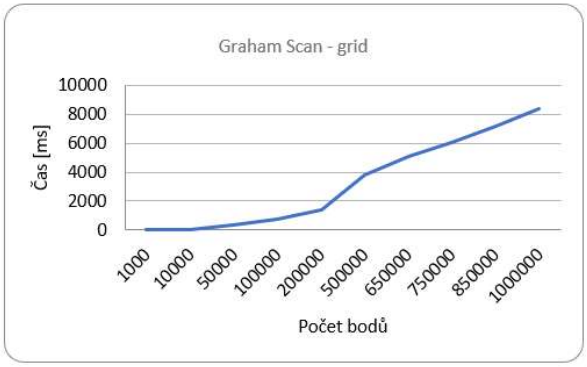
8.4. Graham Scan

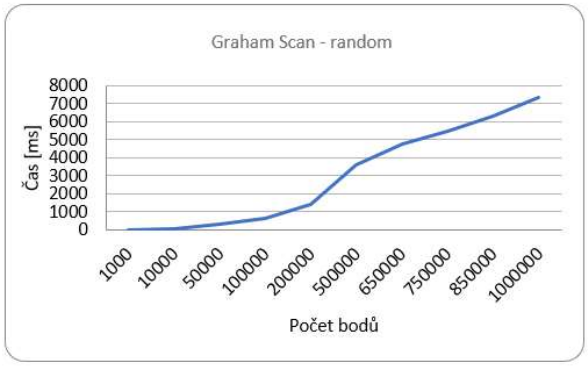
Kružnice										
Počet bodů	**1000**	**10000**	**50000**	**100000**	**200000**	**500000**	**650000**	**750000**	**850000**	**1000000**
Čas [ms]	2	31	239	604	1263	3353	4460	5208	5851	7102
	2	30	236	663	1242	3329	4445	5248	5880	7131
	2	32	234	595	1247	3323	4461	5129	5984	7032
	2	32	236	610	1346	3317	4467	5144	5817	7025
	2	33	238	601	1290	3332	4473	5251	5892	7030
	2	28	237	592	1253	3328	4515	5326	6000	7017
	2	33	240	601	1274	3315	4434	5207	5813	6936
	2	32	239	598	1269	3284	4438	5175	5795	6859
	2	32	243	588	1280	3311	4621	5259	5847	6957
	2	26	235	581	1264	3314	4412	5170	5740	6925
Průměr	2	30,9	237,7	603,3	1272,8	3320,6	4472,6	5211,7	5861,9	7001,4
Rozptyl	0	4,69	6,41	457,61	790,16	283,04	3126,64	3270,81	5929,69	6183,44

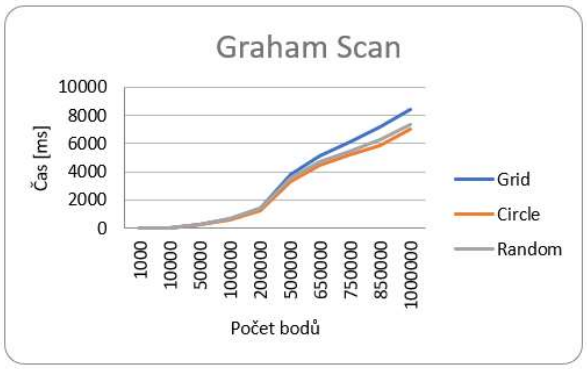
Grid										
Počet bodů	**1000**	**10000**	**50000**	**100000**	**200000**	**500000**	**650000**	**750000**	**850000**	**1000000**
Čas [ms]	4	59	304	677	1409	3833	5036	6078	7171	8409
	5	61	303	715	1395	3806	5089	6053	7172	8351
	4	63	308	710	1405	3795	5176	6094	7170	8419
	4	58	314	703	1433	3734	5082	6069	7115	8451
	5	62	311	706	1403	3832	5094	6109	7169	8388
	4	58	313	703	1397	3836	5081	6119	7212	8366
	4	59	302	704	1395	3790	5142	6084	7145	8379
	5	57	312	726	1406	3789	5071	6121	7167	8359
	4	57	315	741	1376	3803	5094	6049	7147	8436
	4	60	310	701	1399	3862	5100	6129	7174	8417
Průměr	4,3	59,4	309,2	708,6	1401,8	3808	5096,5	6090,5	7164,2	8397,5
Rozptyl	0,21	3,84	20,16	254,24	184,36	1126	1323,25	736,85	563,76	1038,85

Random										
Počet bodů	**1000**	**10000**	**50000**	**100000**	**200000**	**500000**	**650000**	**750000**	**850000**	**1000000**
Čas [ms]	5	57	325	647	1366	3642	4657	5568	6374	7211
	5	60	311	690	1437	3547	4736	5475	6460	7450
	5	62	313	674	1380	3635	4694	5446	6551	7475
	5	67	307	662	1396	3496	4720	5454	6288	7545
	5	59	314	670	1434	3687	4758	5492	6316	7430
	5	56	317	665	1401	3662	4724	5484	6289	7336
	5	61	308	645	1408	3636	4739	5372	6172	7207
	5	56	311	636	1415	3587	4778	5456	6180	7405
	5	59	306	652	1437	3567	4743	5540	6211	7382
	5	55	299	651	1426	3593	4732	5461	6048	7276
Průměr	5	59,2	311,1	659,2	1410	3605,2	4728,1	5474,8	6288,9	7371,7
Rozptyl	0	11,56	43,89	233,36	543,2	3035,96	1010,29	2569,16	19557,5	11483,2



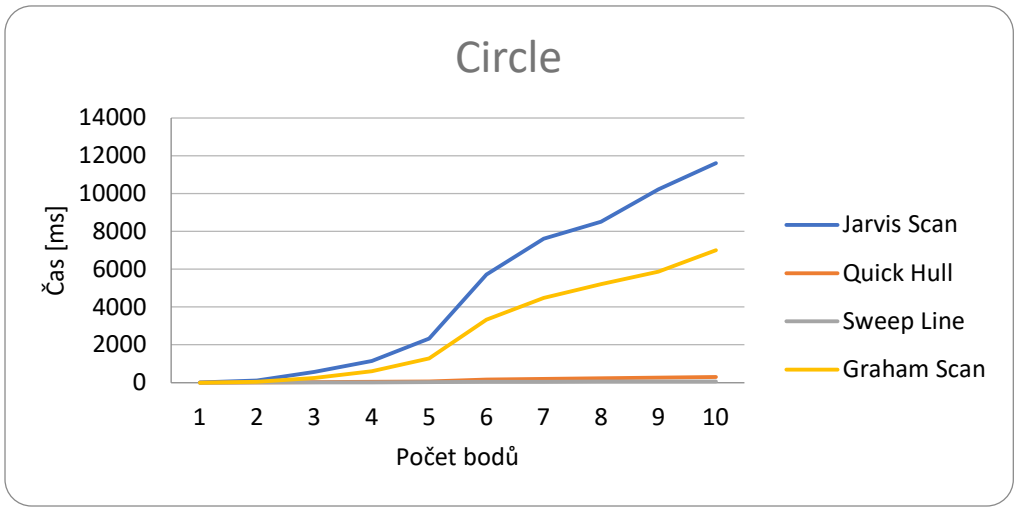


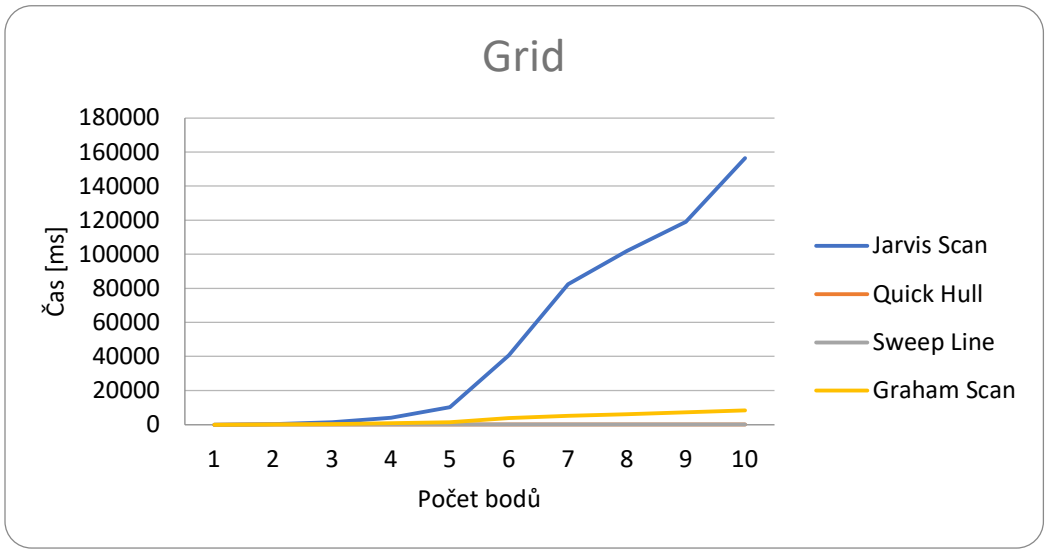


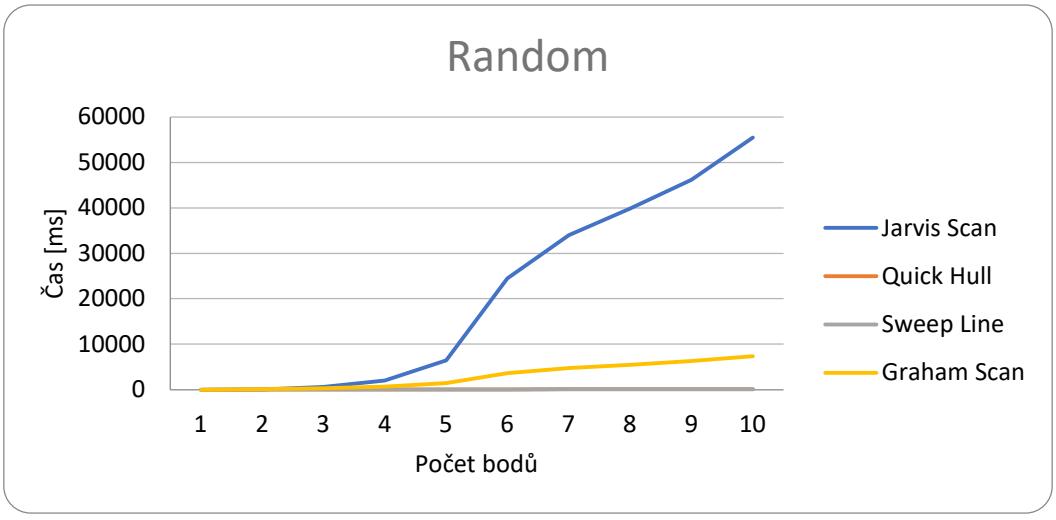


Zde se také ukázala složitost algoritmu Graham Scan $O(n \cdot log\ n)$ jako vhodná aj pro rozsáhlé datasety, ale už méně jako při algoritmech Quick Hull a Sweep Line. Čas při milionu bodech oproti 1000 je při kružnici větší přibližně 3500 násobně, při gridu 2000 násobně a při náhodných bodech 1500 násobně. Z grafů vidět, že od počtu 200000 bodů čas stoupá, jako ve všech algoritmech. Jako nejlepší možnost je při Graham Scan algoritmu jeví náhodné vykreslení bodů.

8.5. Porovnání všech algoritmů







9. Zhodnocení dosažených výsledků a závěr

Byla vytvořená funkční aplikace nevykazující žádné chyby, které by neumožňovali její přeložení a spuštění. Aplikace počítá konvexní obálku pro čtyři algoritmy – Jarvis Scan, Quick Hull, Sweep Line a Graham Scan. Umožňuje také automatické generování kružnice, gridu, náhodných bodů a také tvorbu striktně konvexní obálky.
Z testování jednotlivých algoritmů jsme zjistili, že časově nejnáročnější je algoritmus Jarvis Scan. Naopak jako nejrychlejší se ukázali Quick Hull a Sweep line.
Výpočet konvexní obálky bodů generovaných v pravidelné mřížce pro algoritmus Graham Scan nepracuje dobře. Výpočet nezahrne bod v pravém horní rohu. Také tento algoritmus padá při větším množství bodů (když se při výpočtu klikne myší někam mimo okno aplikace).

9.1. Řešené bonusové úlohy

Řešenými bonusovými úlohami bylo naprogramováni algoritmu Graham Scan, konstrukce striktně konvexních obálek a automatické generování množin bodů ve tvare kružnice, gridu a náhodně umístněných bodů.

9.2. Možné vylepšení

Do aplikace by se ještě mohlo přidat jiné generování bodů (star-shaped, obdélník, elipsa…). Také podle bonusové úlohy konstrukce Minimum Area Enclosing box – funkce pro hledání hlavních směrů konvexních obálek (budov).

10. Seznam obrázků