Geometria Computacional

Envoltório Convexo

Prof. Edson Alves

2019

Faculdade UnB Gama

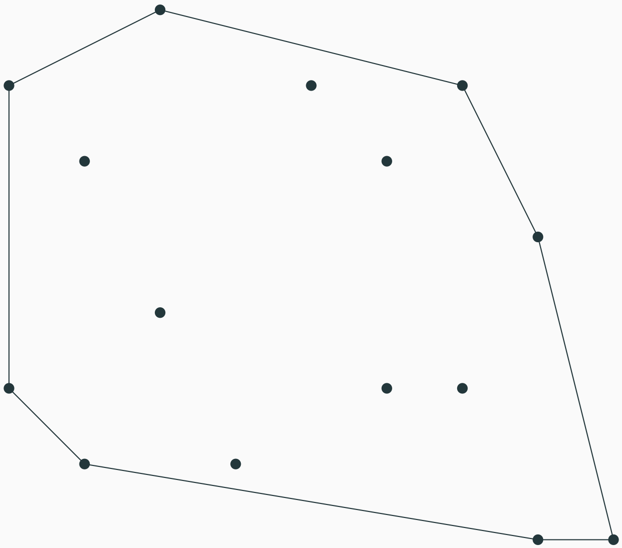
Sumário

Definição

Envoltório convexo

- Dado um conjunto de N pontos P, o envoltório convexo $C_H(P)$ de P (*convex hull*) é o menor polígono convexo tal que cada ponto de P ou pertence ao interior de $C_H(P)$ ou é um de seus vértices
- O termo menor na definição acima se refere à menor área
- O envoltório convexo não é único, pois não impõem restrição na orientação do polígono
- Existem vários algoritmos para se determinar o envoltório convexo
- O mais conhecido é o algoritmo de Graham
- Além deles, outros dois algoritmos importantes são a cadeia monótona de Andrew e a marcha de Jarvis
- Como os vértices de $C_H(P)$ são pontos de P, a essência dos algoritmos é determinar, para cada ponto de P, se ele pertence ou não ao $C_H(P)$

Exemplo de envoltório convexo



Algoritmo de Graham

Algoritmo de Graham

- O algoritmo de Graham (*Graham Scan*, no original), foi proposto por Ronald Graham em 1972
- Ele iniciamente ordena todos os N pontos de P de acordo com o ângulo que eles foram com um ponto pivô fixado previamente
- A escolha padrão para o pivô é o ponto de menor coordenada y
- Caso exista mais de um ponto com coordenada y mínima, escolhe-se o de maior coordenada x dentre eles
- Se P é armazenado em um vetor, o algoritmo pode ser simplificado movendo-se o pivô para a primeira posição

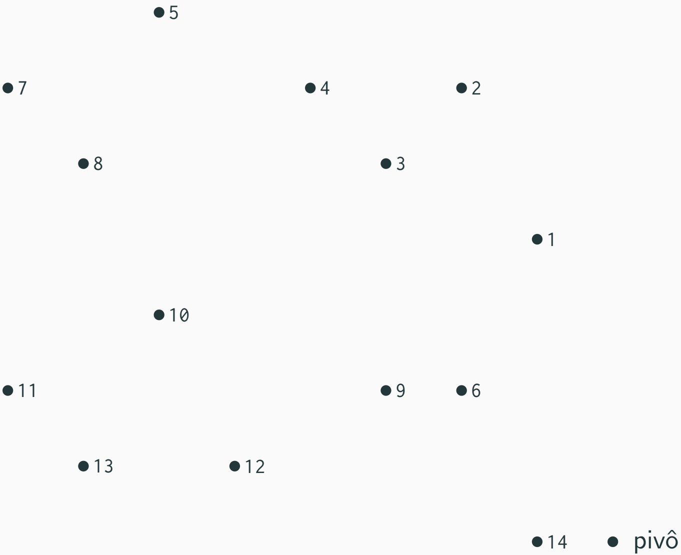
Implementação da escolha do pivô

```
template<typename T>
class GrahamScan
{
private:
    static Point<T> pivot(vector<Point<T>>& P)
    {
        size_t idx = 0;

        for (size_t i = 1; i < P.size(); ++i)
            if (P[i].y < P[idx].y or
                (equals(P[i].y, P[idx].y) and P[i].x > P[idx].x))
                    idx = i;

        swap(P[0], P[idx]);

        return P[0];
    }

```

Ordenação dos pontos de acordo com o ângulo

- Para realizar a ordenação dos pontos é preciso definir um operador booleano que receba dois pontos P e Q e retorne verdadeiro se P antecede Q de acordo com a ordenação proposta
- Como é necessário o conhecimento do pivô para tal ordenação, há três possibilidades para a implementação deste operador:
    1. implementar o operator < da classe `Point`, tornando o pivô um membro da classe para que o operador tenha acesso a ele;
    2. tornar o pivô uma variável global;
    3. usar uma função lambda no terceiro parâmetro da função `sort()`, capturando o pivô por referência ou cópia
- O ângulo que o vetor diferença entre o vetor-posição do pivô e o vetor posição de um ponto do conjunto P faz com o eixo-x positivo pode ser obtido através da função `atan2()` da biblioteca `math.h` da linguagem C/C++

Exemplo de ordenação por ângulo



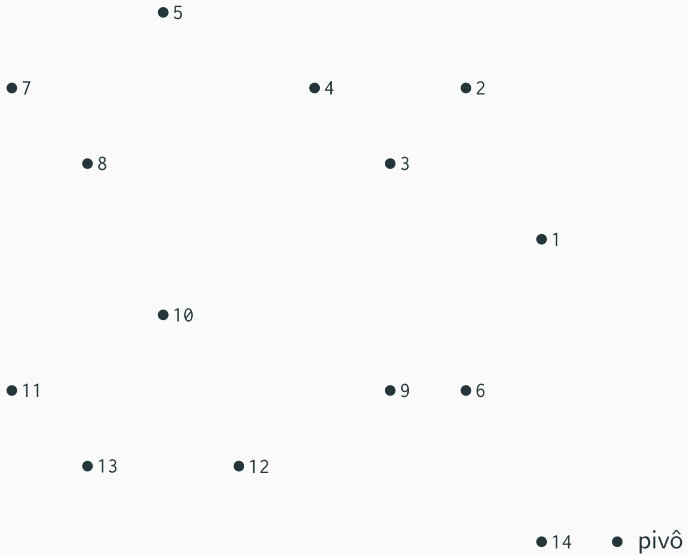
Implementação da rotina de ordenação dos pontos

```
    static void sort_by_angle(vector<Point<T>>& P)
    {
        auto P0 = pivot(P);

        sort(P.begin() + 1, P.end(),
            [&](const Point<T>& A, const Point<T>& B) {
                // pontos colineares: escolhe-se o mais próximo do pivô
                if (equals(D(P0, A, B), 0))
                    return A.distance(P0) < B.distance(P0);

                auto alfa = atan2(A.y - P0.y, A.x - P0.x);
                auto beta = atan2(B.y - P0.y, B.x - P0.x);

                return alfa < beta;
            }
        );
    }

```

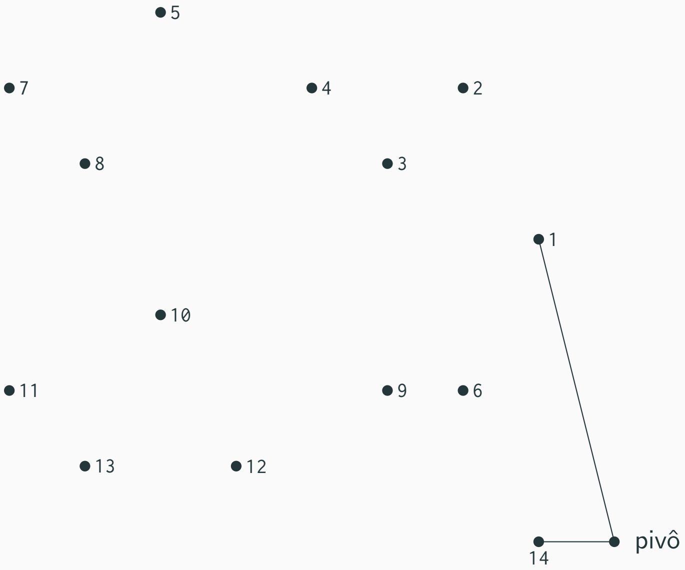
Identificação do envoltório convexo

- Após a ordenação dos pontos, o algoritmo procede empilhando três pontos de P: inicialmente os pontos cujos índices são $n-1, 0$ e 1
- O invariante a ser mantido é que os três elementos do topo de pilha estão em sentido anti-horário $(D > 0)$
- Para cada um dos demais pontos Q_i de P, com $i = 2, 3, \ldots, n-1$, verifica-se se este ponto mantem o sentido anti-horário com os dois elementos do topo da pilha
- Em caso afirmativo, o ponto é inserido na pilha
- Caso contrário, remove-se o topo da pilha e se verifica o invariante para Q_i novamente
- Como cada ponto é ou inserido ou removido uma única vez, este processo tem complexidade $O(N)$, e o algoritmo como um todo tem complexidade $O(N \log N)$, devido à ordenação

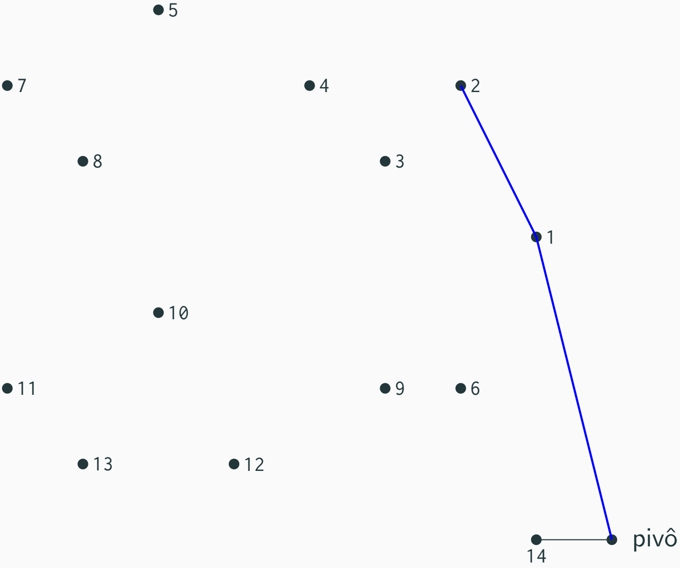
Visualização do algoritmo de Graham



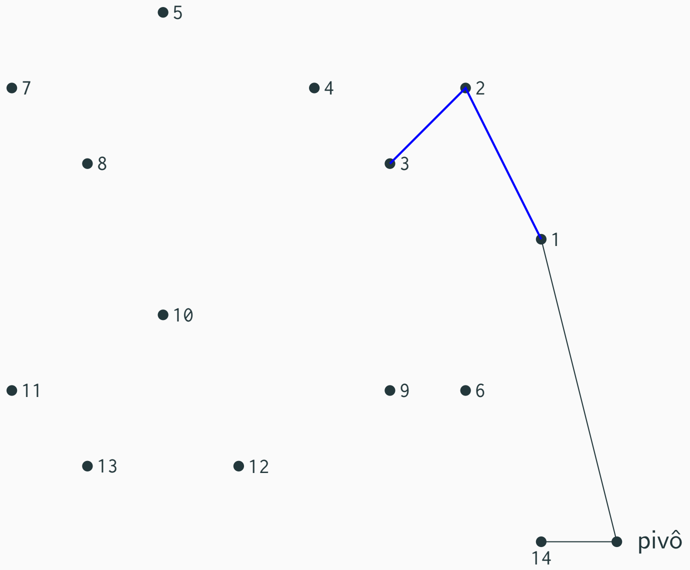
Visualização do algoritmo de Graham



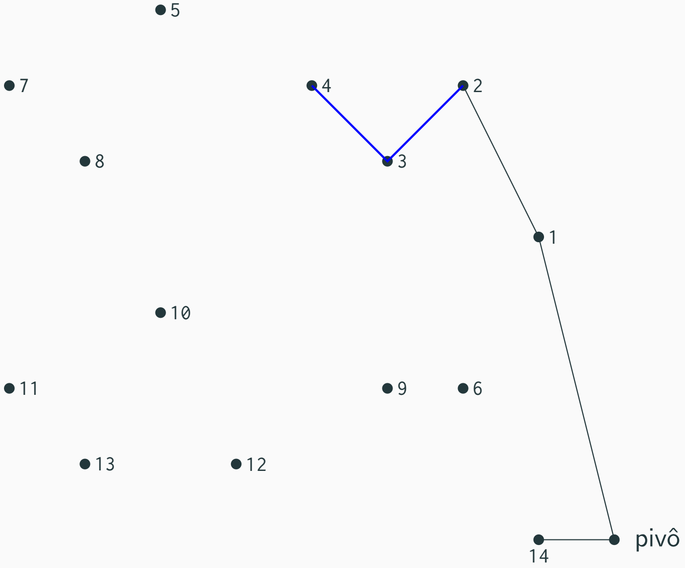
Visualização do algoritmo de Graham



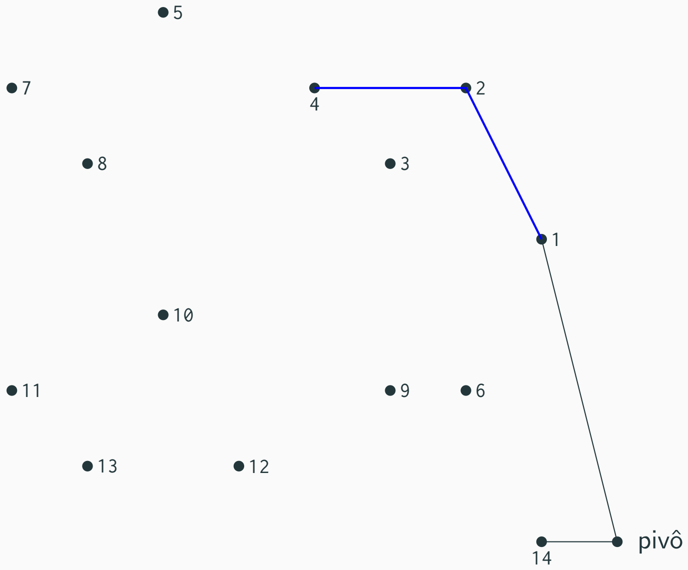
Visualização do algoritmo de Graham



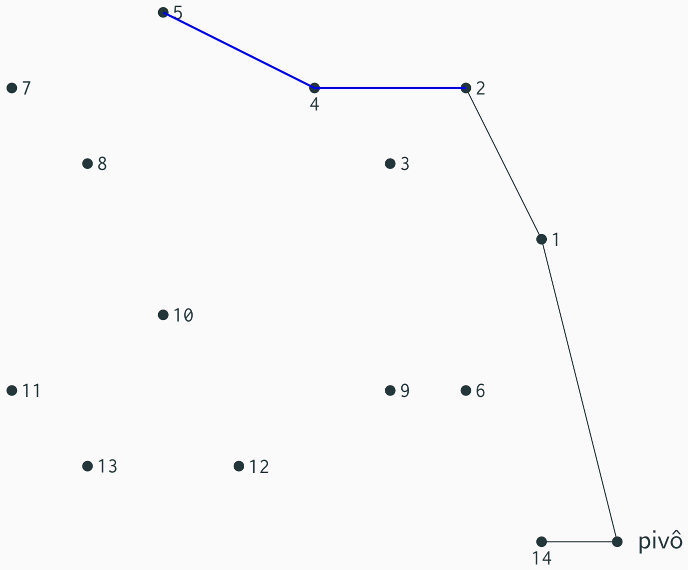
Visualização do algoritmo de Graham



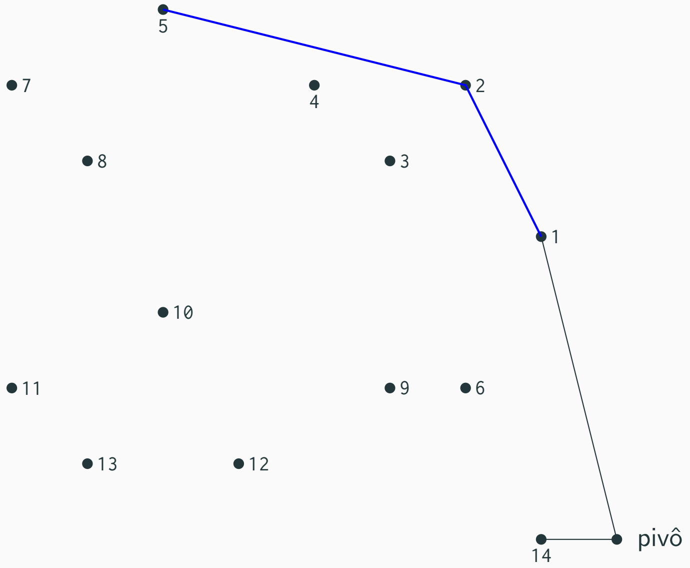
Visualização do algoritmo de Graham



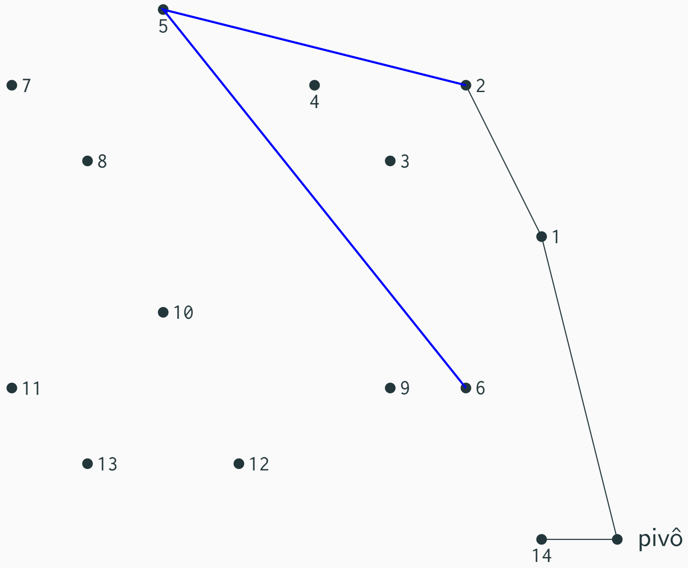
Visualização do algoritmo de Graham



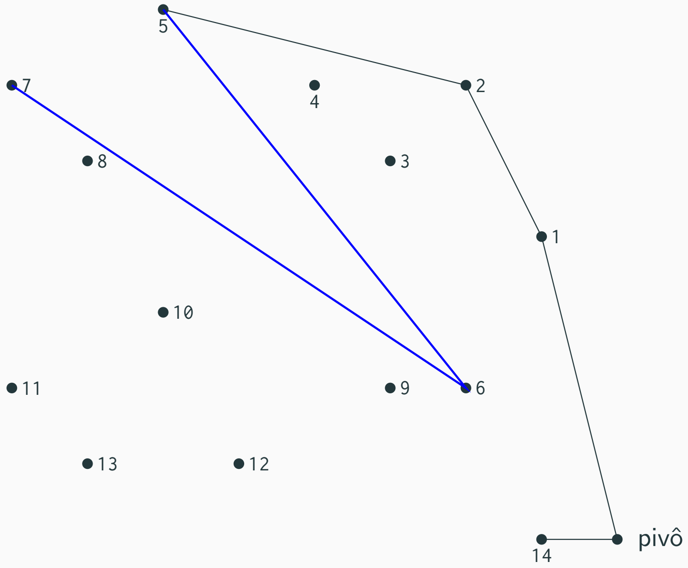
Visualização do algoritmo de Graham



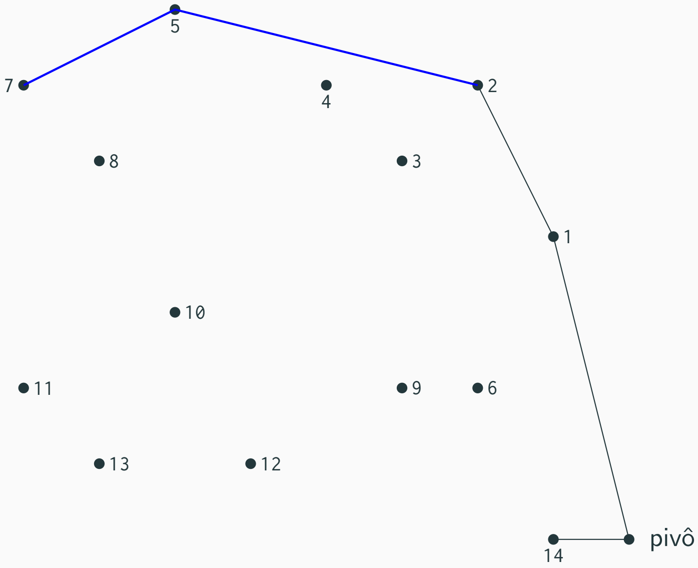
Visualização do algoritmo de Graham



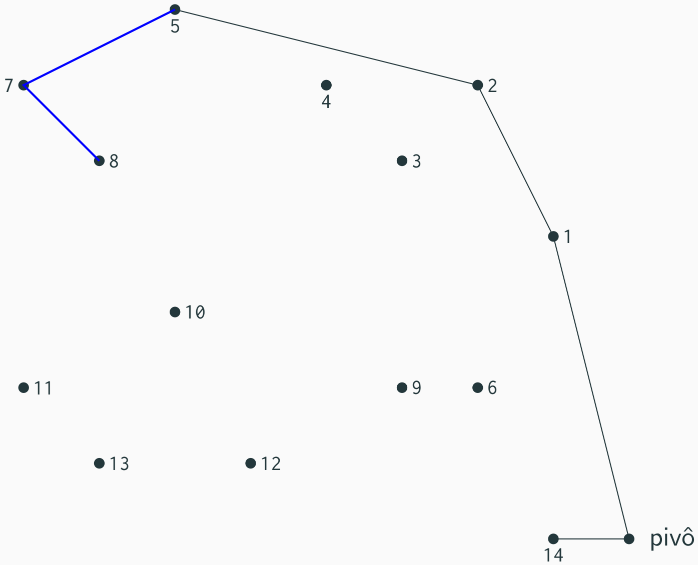
Visualização do algoritmo de Graham



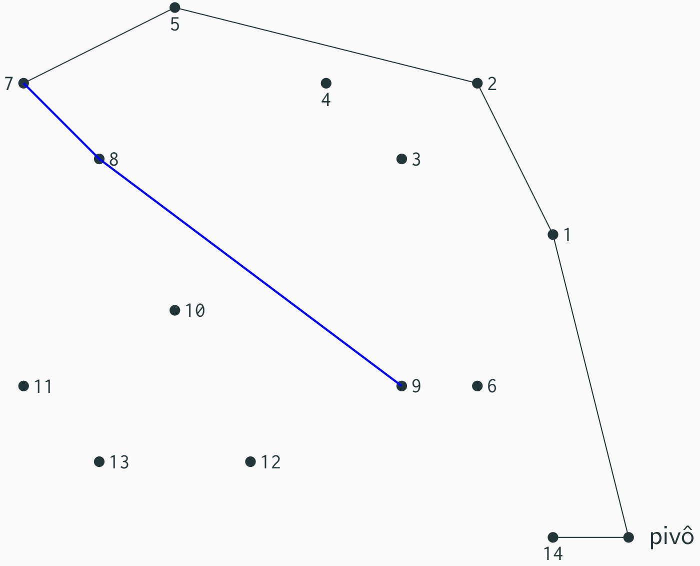
Visualização do algoritmo de Graham



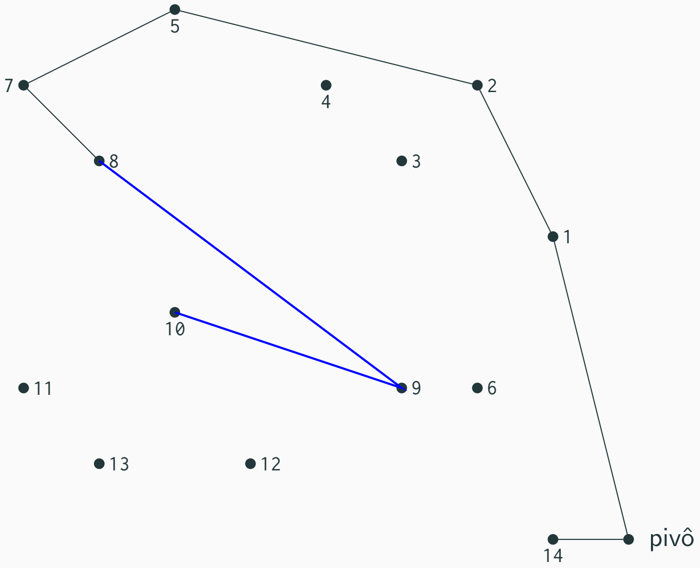
Visualização do algoritmo de Graham



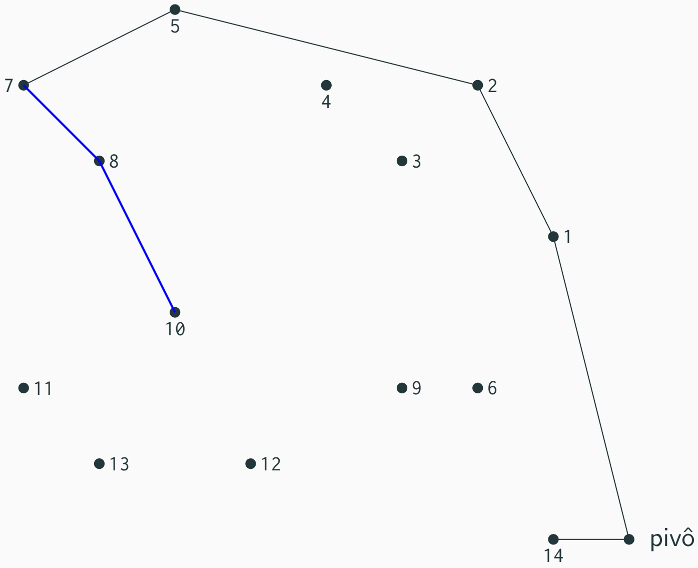
Visualização do algoritmo de Graham



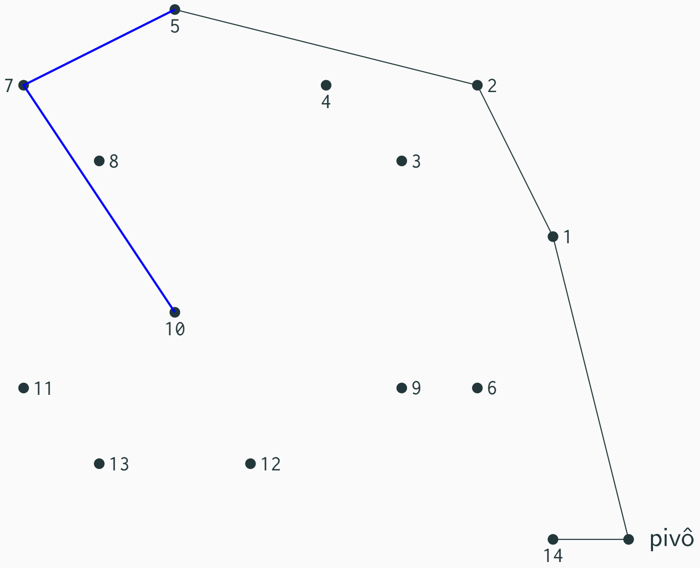
Visualização do algoritmo de Graham



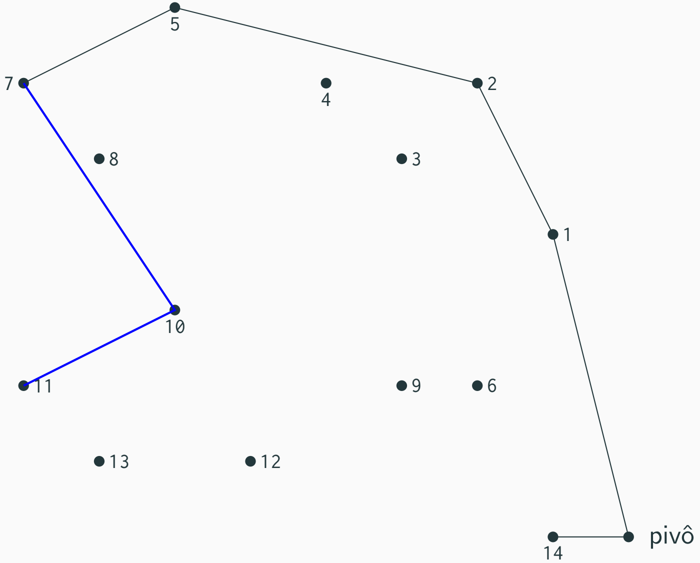
Visualização do algoritmo de Graham



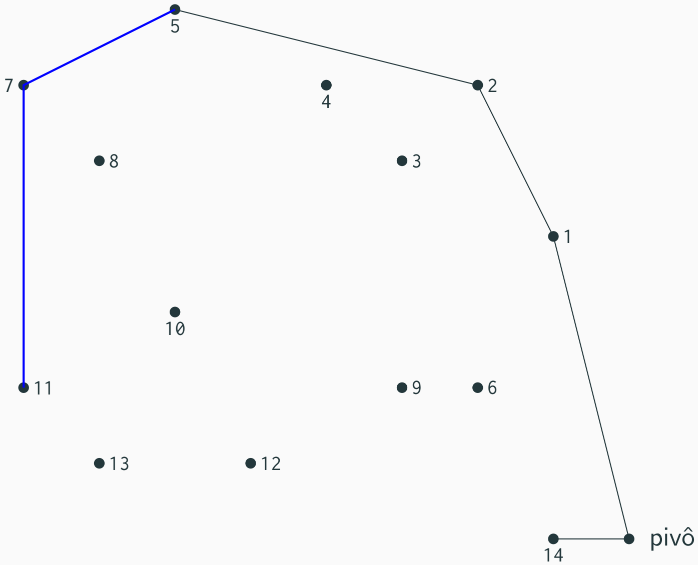
Visualização do algoritmo de Graham



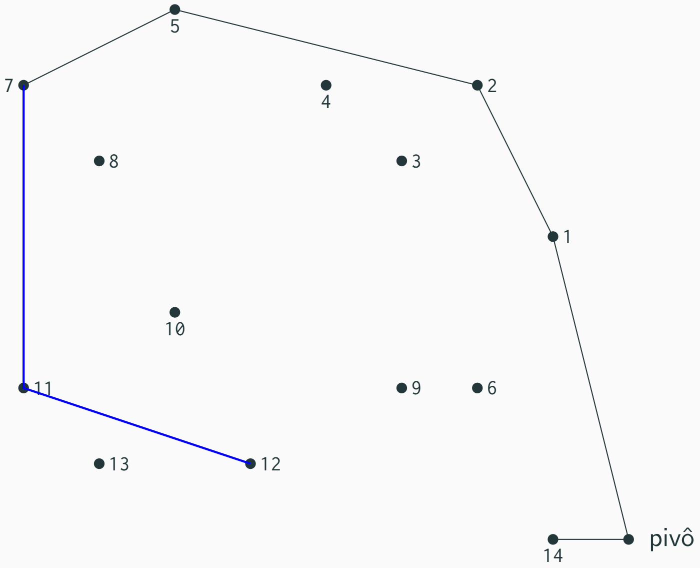
Visualização do algoritmo de Graham



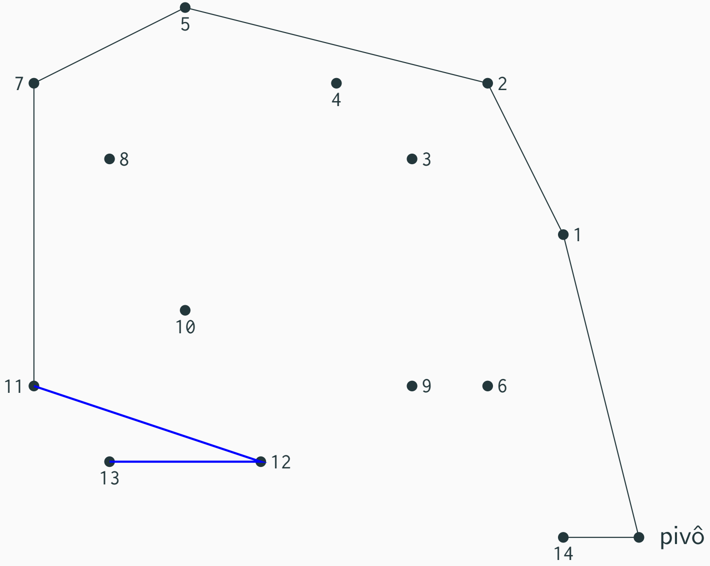
Visualização do algoritmo de Graham



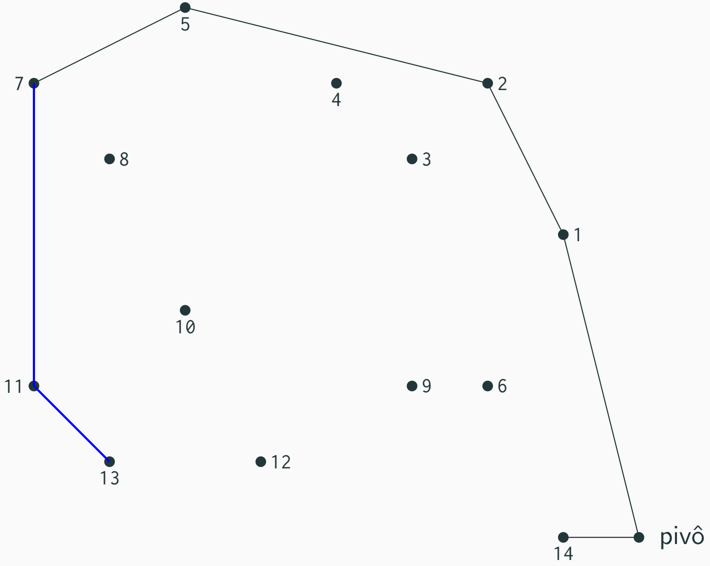
Visualização do algoritmo de Graham



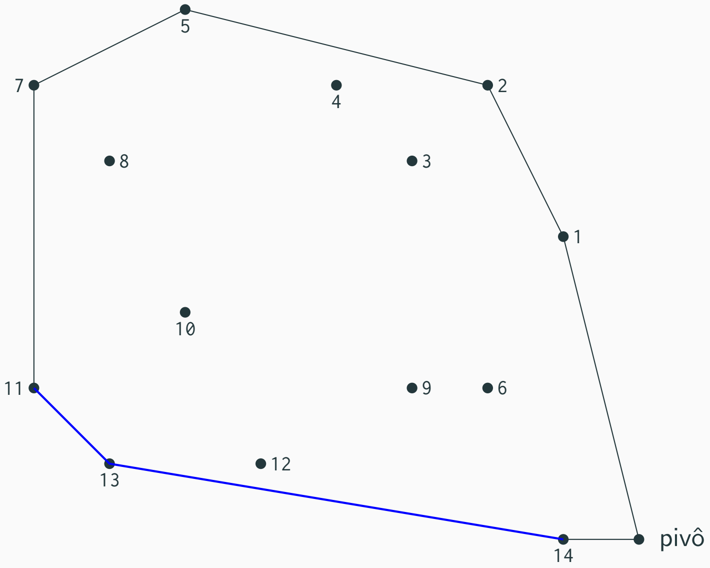
Visualização do algoritmo de Graham



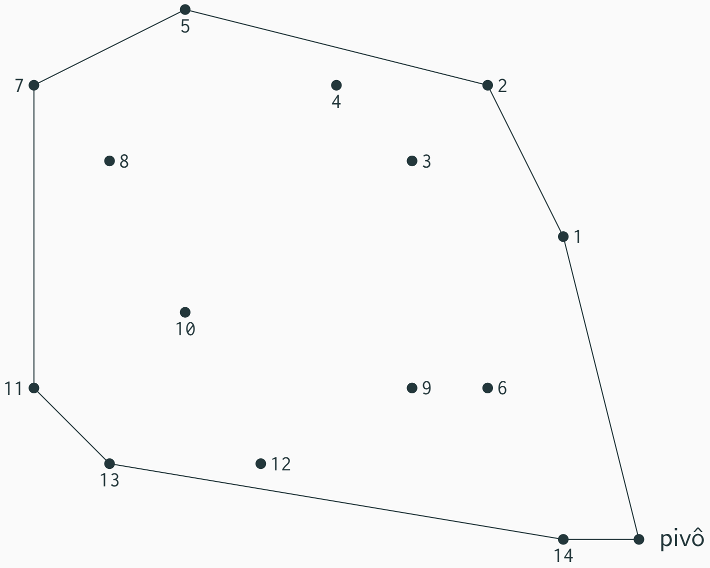
Visualização do algoritmo de Graham



Visualização do algoritmo de Graham



Visualização do algoritmo de Graham



Implementação da rotina de envoltório convexo

```
public:
    static vector<Point<T>> convex_hull(const vector<Point<T>>& points)
    {
        vector<Point<T>> P(points);
        auto N = P.size();

        // Corner case: com 3 vértices ou menos, P é o próprio convex hull
        if (N <= 3)
            return P;

        sort_by_angle(P);

        vector<Point<T>> ch;
        ch.push_back(P[N - 1]);
        ch.push_back(P[0]);
        ch.push_back(P[1]);

        size_t i = 2;

```

Implementação da rotina de envoltório convexo

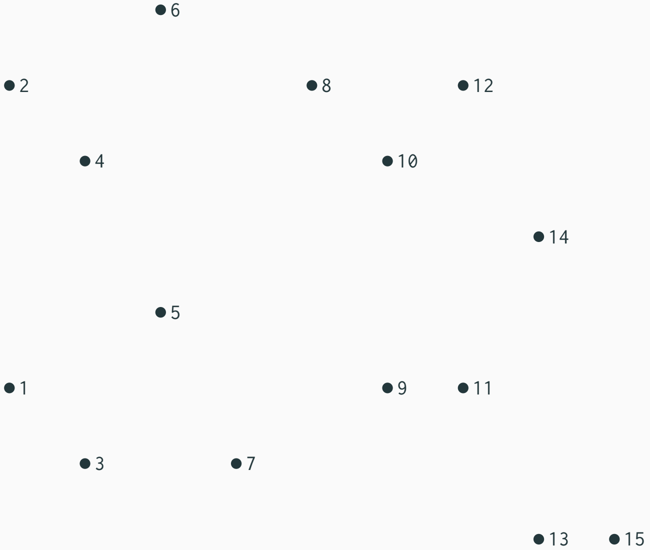
```
        while (i < N)
        {
            auto j = ch.size() - 1;

            if (D(ch[j - 1], ch[j], P[i]) > 0)
                ch.push_back(P[i++]);
            else
                ch.pop_back();
        }

        // O envoltório é um caminho fechado: o primeiro ponto é igual
        // ao último
        return ch;
    }
};
```

Cadeia monótona de Andrew

Algoritmo de Andrew

- O algoritmo conhecido como cadeia monótona de Andrew (*Andrew's Monotone Chain Algorithm*, no original) é uma alternativa ao algoritmo de Graham para a geração do envoltório convexo
- Este algoritmo foi proposto por Andrew em 1979
- A complexidade é a mesma do algoritmo de Graham: $O(N \log N)$
- Ele constrói o envoltório em duas partes: a parte superior (*upper hull*) e a parte inferior (*lower hull*)
- Os pontos são ordenados por coordenada x e, em caso de empate, por coordenada y

Exemplo de ordenação por coordenadas



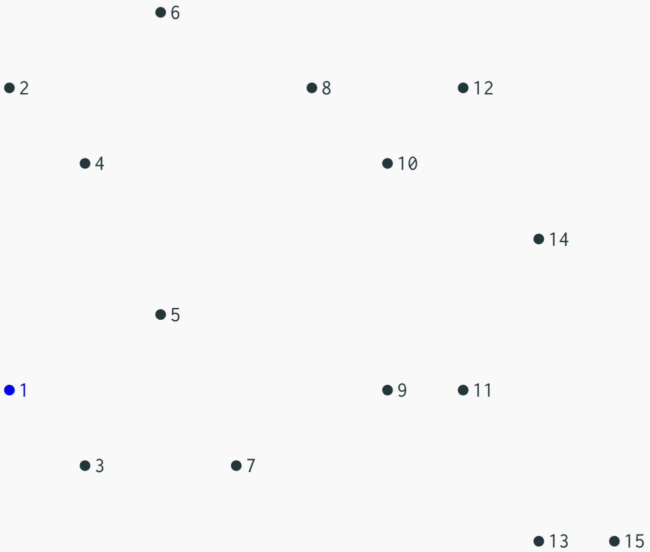
Implementação da rotina de comparação de pontos

```cpp
#include <bits/stdc++.h>

using namespace std;

template<typename T>
struct Point
{
    T x, y;

    bool operator<(const Point& P) const
    {
        return x == P.x ? y < P.y : x < P.x;
    }
};

template<typename T>
T D(const Point<T>& P, const Point<T>& Q, const Point<T>& R)
{
    return (P.x * Q.y + P.y * R.x + Q.x * R.y) -
           (R.x * Q.y + R.y * P.x + Q.x * P.y);
}
```

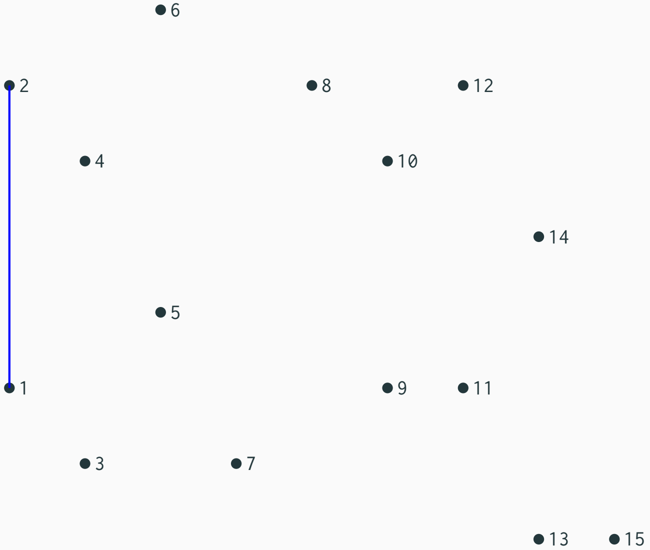
Geração do envoltório convexo

- O envoltório convexo é gerado de forma semelhante ao procedimento usado no algoritmo de Graham
- O ponto de partida é o ponto mais à esquerda, com menor coordenada y
- O *lower hull* é gerado empilhando os pontos de acordo com a ordenação, desde que o novo ponto e os dois últimos elementos da pilha mantenham a orientação anti-horária, ou que a pilha tenham menos do que dois elementos
- Para gerar o *upper hull*, é preciso começar do ponto mais à direita, com maior coordenada y
- A rotina é idêntica à usado no *lower hull*: basta processar os pontos do maior para o menor, de acordo com a ordenação
- Ao final as duas partes devem ser unidas
- O ponto final do *lower hull* deve ser descartado, uma vez que é idêntico ao ponto inicial do *upper hull*

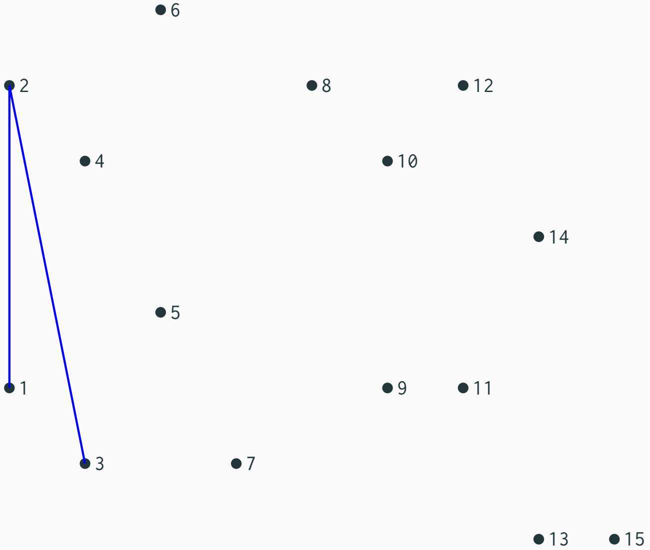
Geração do lower hull



Geração do lower hull

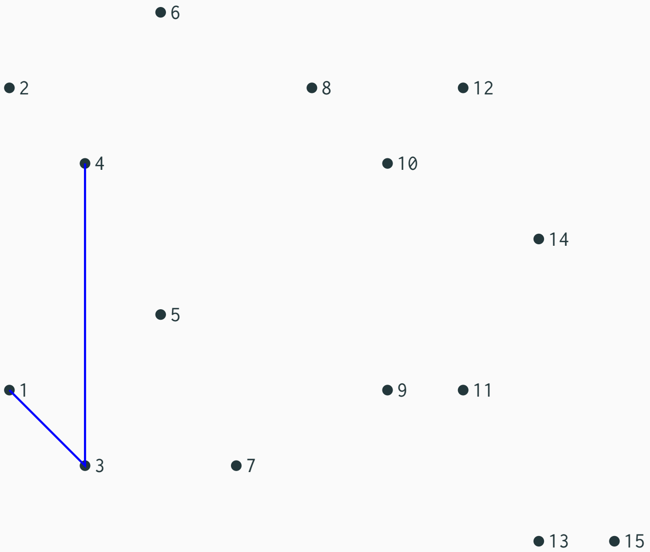


Geração do lower hull

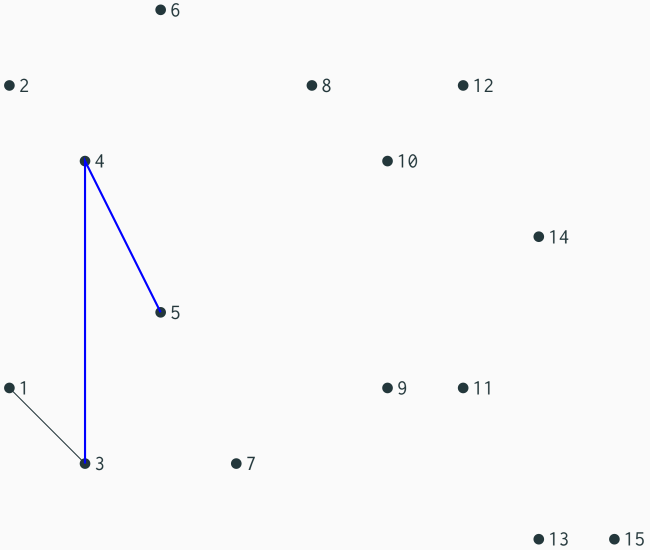


Geração do lower hull

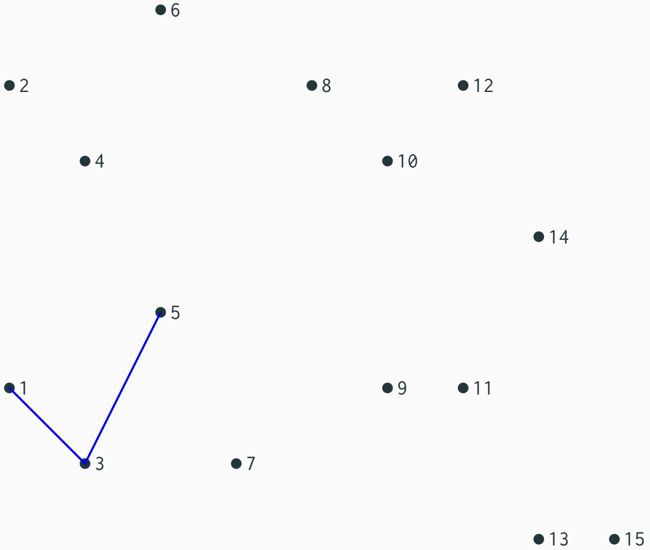
Geração do lower hull



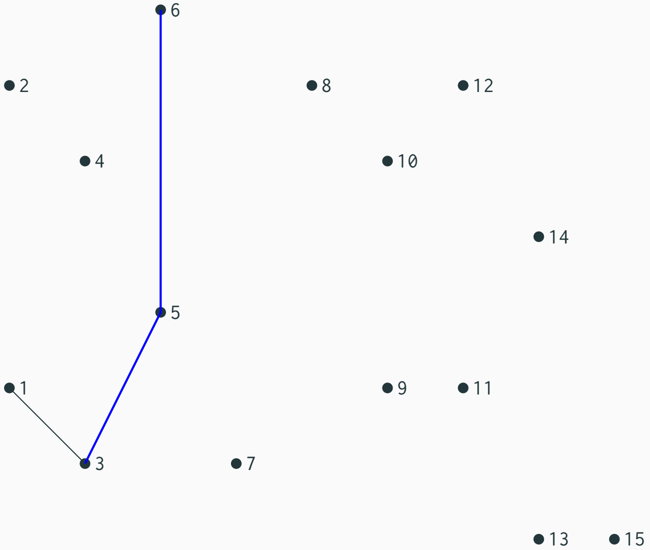
Geração do lower hull



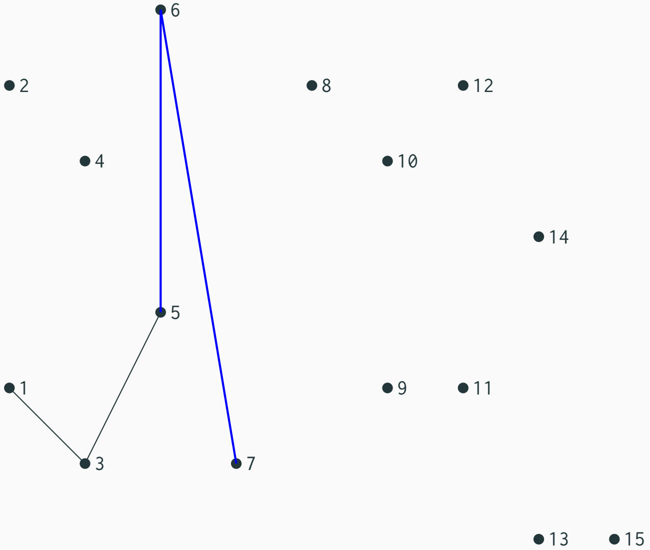
Geração do lower hull



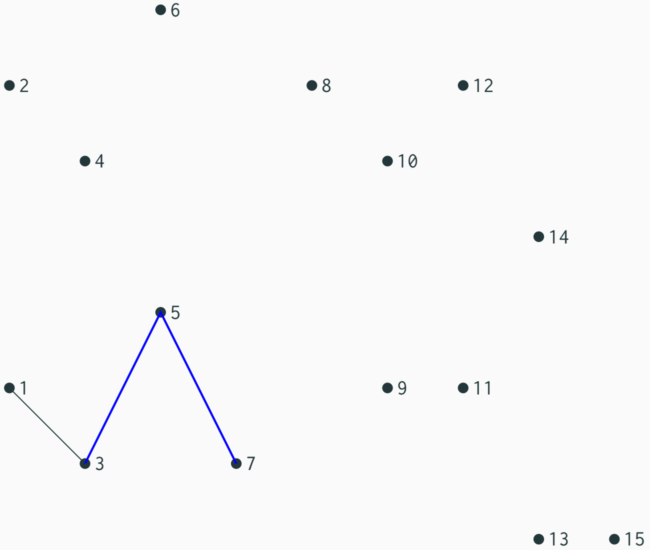
Geração do lower hull



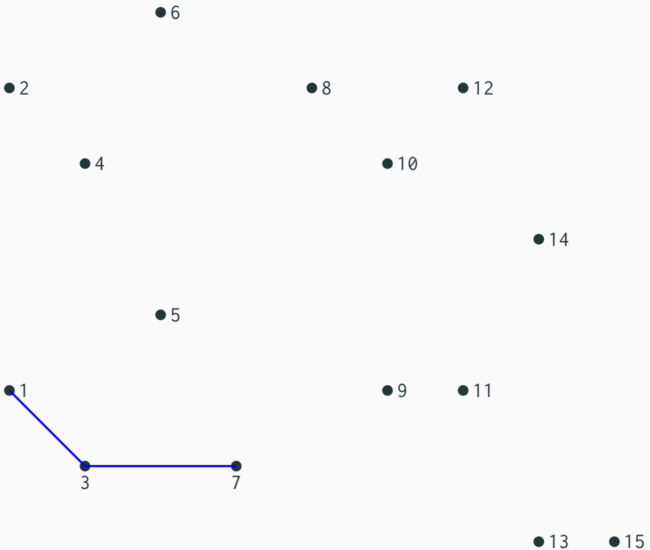
Geração do lower hull



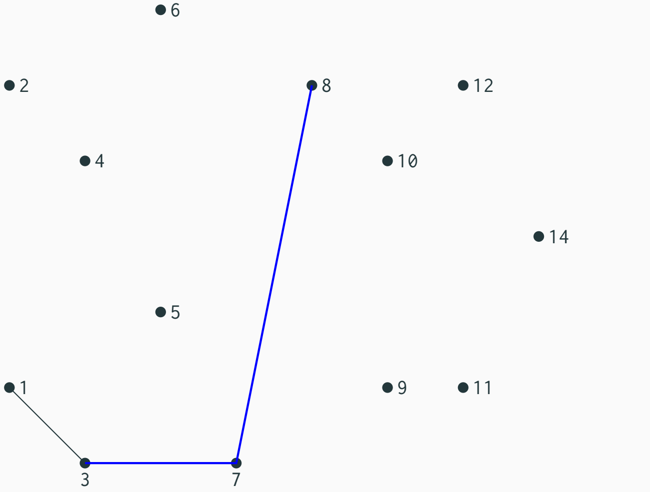
Geração do lower hull



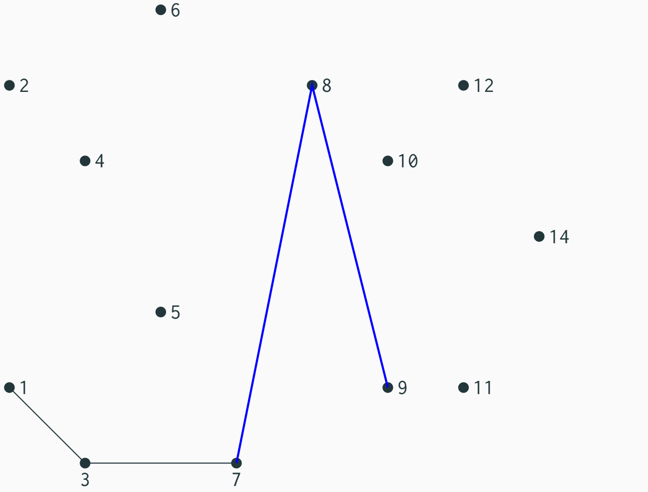
Geração do lower hull



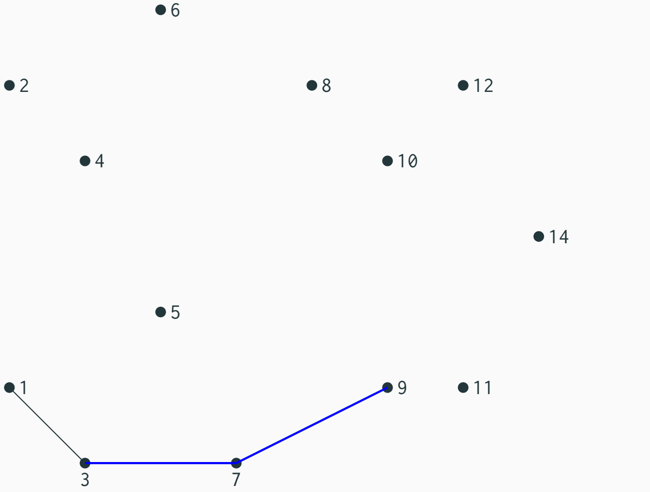
Geração do lower hull



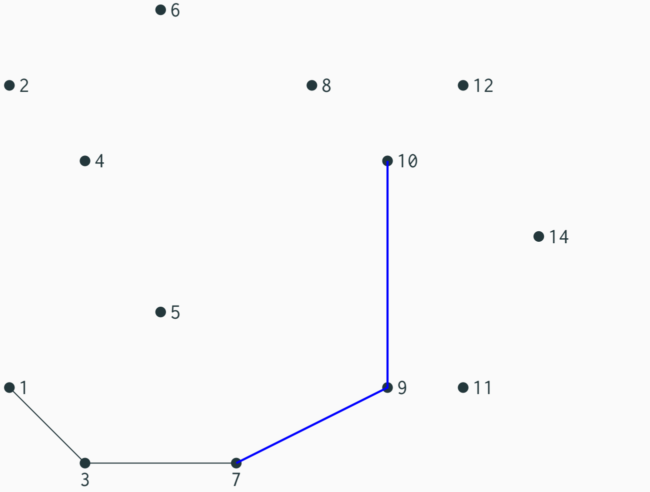
Geração do lower hull



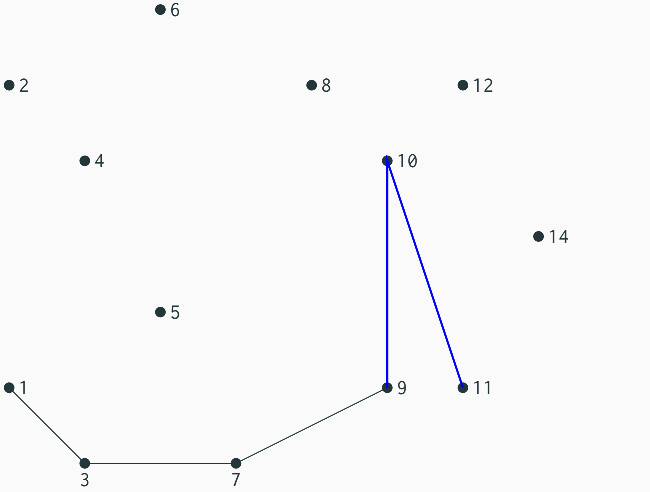
Geração do lower hull



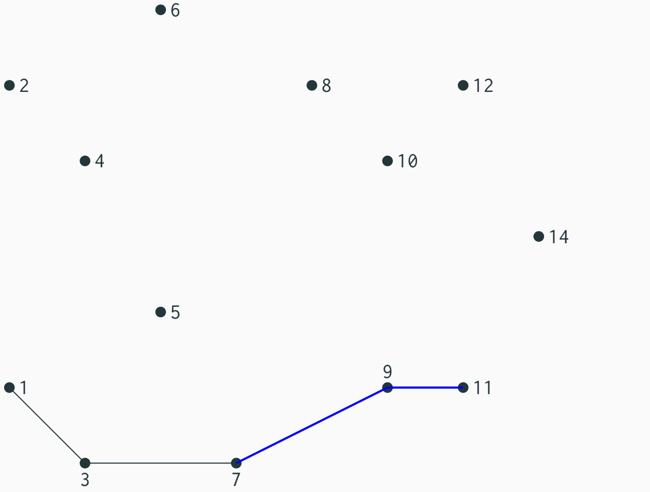
Geração do lower hull



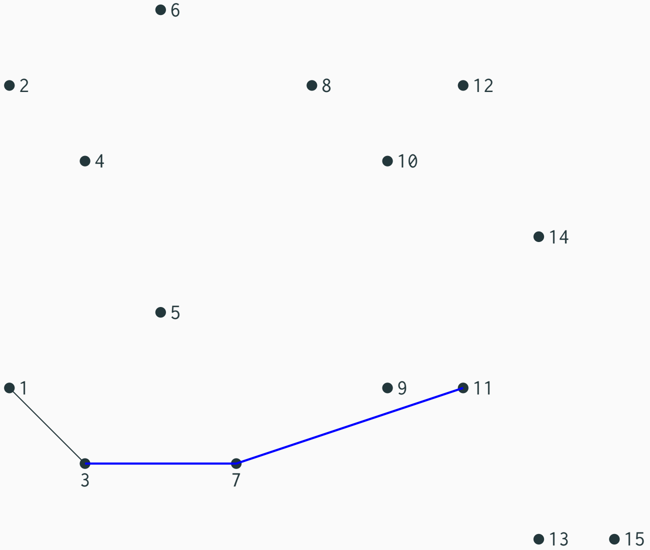
Geração do lower hull



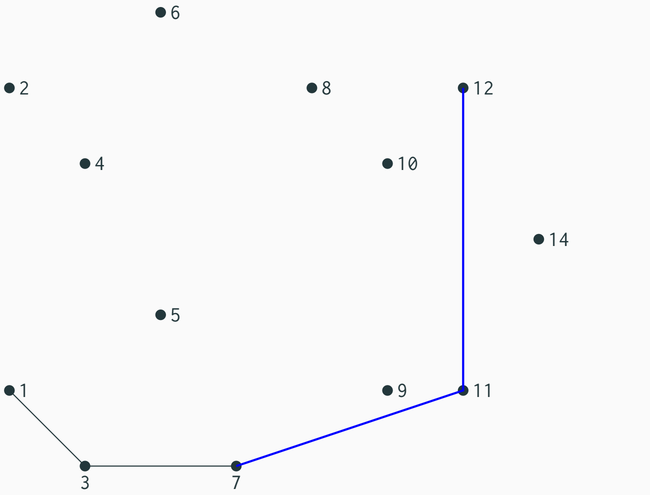
Geração do lower hull



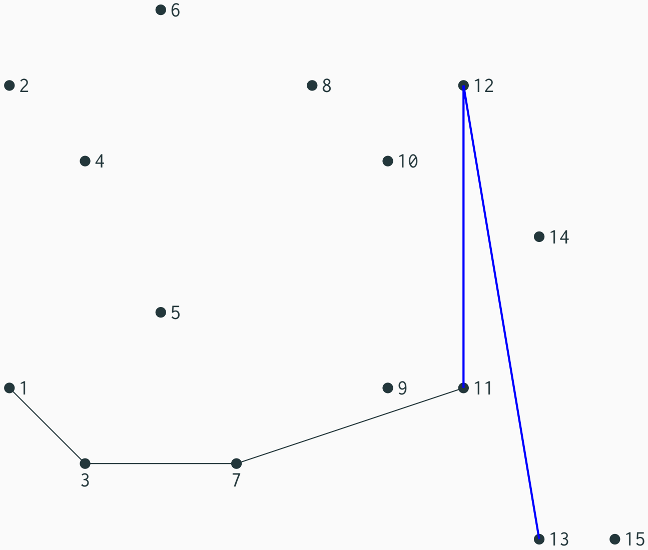
Geração do lower hull



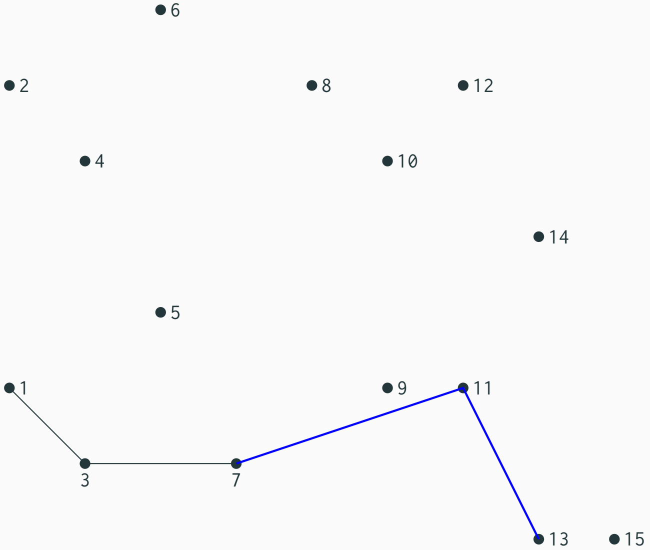
Geração do lower hull



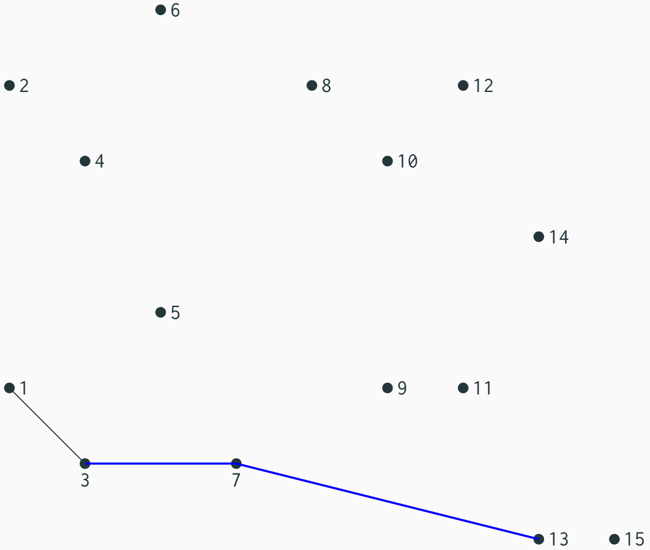
Geração do lower hull



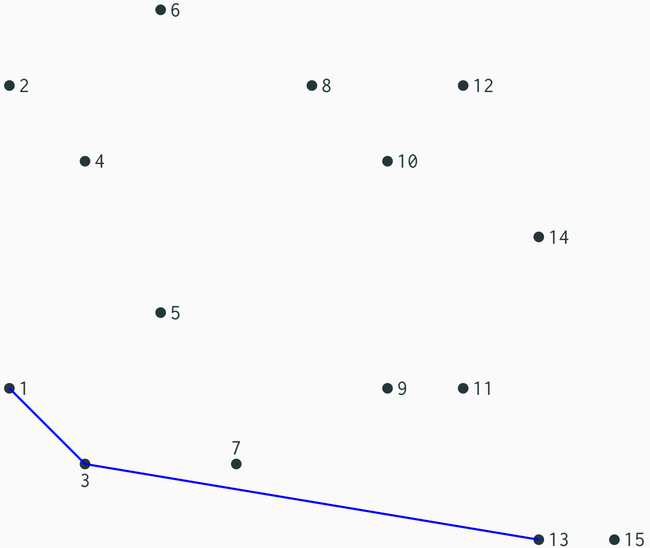
Geração do lower hull



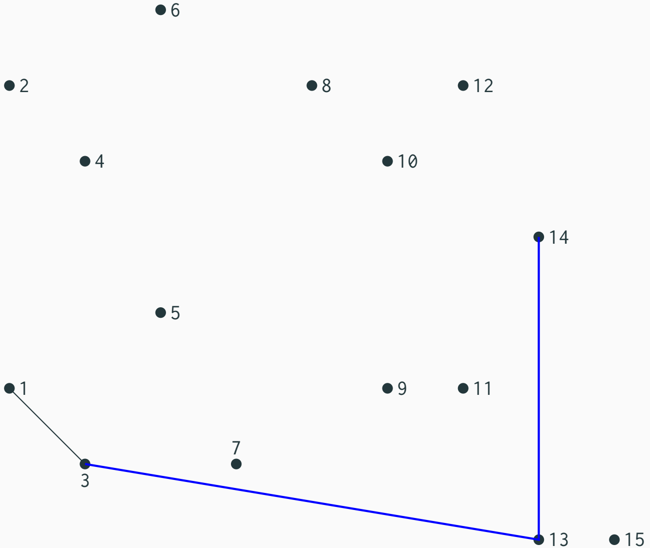
Geração do lower hull



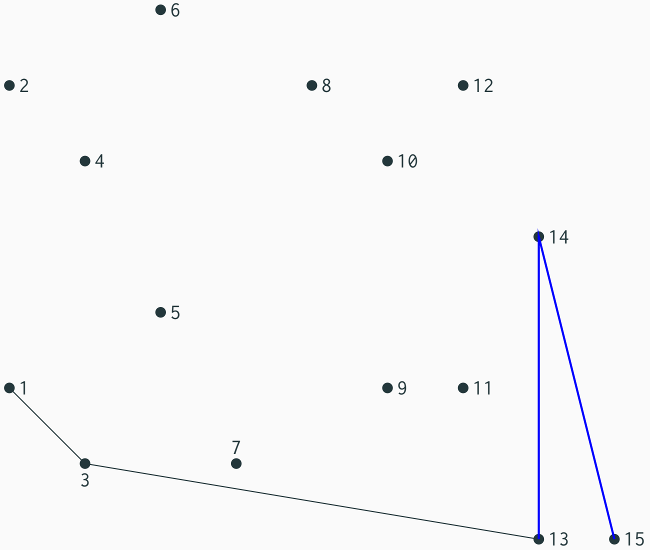
Geração do lower hull



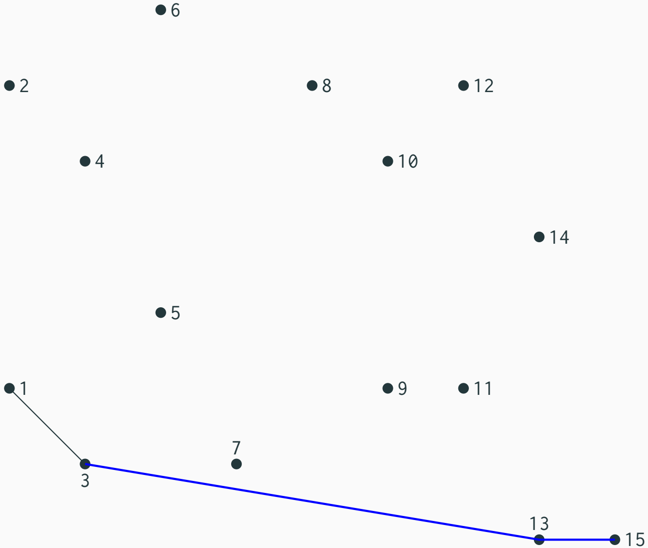
Geração do lower hull



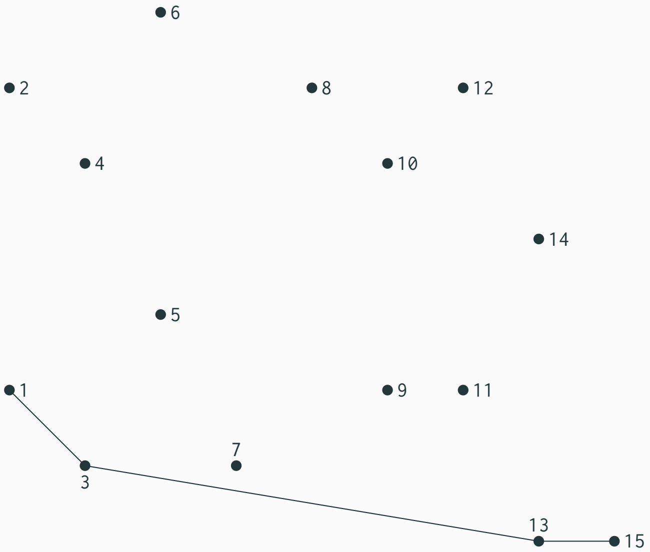
Geração do lower hull



Geração do lower hull



Geração do lower hull



Implementação da geração do envoltório convexo

```cpp
template<typename T>
vector<Point<T>> monotone_chain(const vector<Point<T>>& points)
{
    vector<Point<T>> P(points);

    sort(P.begin(), P.end());

    vector<Point<T>> lower, upper;

    for (const auto& p : P)
    {
        auto size = lower.size();

        while (size >= 2 and D(lower[size - 2], lower[size - 1], p) <= 0)
        {
            lower.pop_back();
            size = lower.size();
        }

        lower.push_back(p);
    }
```

Implementação da geração do envoltório convexo

```
    reverse(P.begin(), P.end());

    for (const auto& p : P)
    {
        auto size = upper.size();

        while (size >= 2 and D(upper[size - 2], upper[size - 1], p) <= 0)
        {
            upper.pop_back();
            size = upper.size();
        }

        upper.push_back(p);
    }

    lower.pop_back();
    lower.insert(lower.end(), upper.begin(), upper.end());

    return lower;
}

```

Referências

1. **ANDREW**, A. M. *Another Efficient Algorithm for Convex Hulls in Two Dimensions*. Information Processing Letters vol. 9, pg. 216-219, 1979.
2. **DE BERG**, Mark. *Computational Geometry: Algorithms and Applications*, Springer, 3rd edition, 2008.
3. **GRAHAM**, R. L. *An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set*. Information Processing Letters vol. 1 (4), pg. 132-133, 1972.
4. **HALIM**, Felix; **HALIM**, Steve. *Competitive Programming 3*, 2010.
5. **LAAKSONEN**, Antti. *Competitive Programmer's Handbook*, 2018 (*Open Access*).
6. **O'ROURKE**, Joseph. *Computational Geometry in C*, Cambridge University Press, 2nd edition, 1998.
7. Wikipedia. Graham scan, acesso em 09/06/2019.
8. Wikipedia. Convex hull algorithms, acesso em 10/05/2019.