Convex Hull: Jarvis March, Graham Scan y Algoritmo de Chan.

Alvarado Contreras Rebeca.
Universidad Autónoma de San Luis Potosí, Facultad de Ciencias.
Ingeniería Física, Cómputo de alto rendimiento.
Proyecto final: Convex Hull.

***Objetivos*—Implementar diferentes algoritmos (*Jarvis March*, *Graham Scan*, *Algoritmo de Chan*) para calcular la envolvente convexa de un conjunto de puntos.**

***Palabras clave*—*Convex Hull*, *Jarvis March*, *Graham Scan*, *Algoritmo de Chan*.**

I. Introducción

El cálculo de la envolvente convexa o *convex hull* es un problema principalmente de geometría computacional. De forma simple, la envolvente convexa de un conjunto de puntos es la región "sólidaz cerrada que incluye todos los puntos en su interior con perímetro mínimo. Puede ser pensado, en el caso 2D, como si se tuviera un conjunto de clavos saliendo del plano, y se pusiera una banda elástica alrededor de estos, de tal manera que encierra a todos los clavos usando la longitud mínima (véase Figura 1).

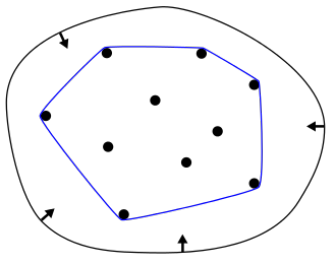


Figura 1: Visualización de la envolvente convexa en 2D.

Los principales algoritmos para calcular la envolvente convexa son:

- Jarvis March o *Gift wrapping* con $O(nh)$.
- Graham Scan con $O(n \log n)$.
- *Divide and conquer* con $O(n \log n)$.
- *Quickhull* con $O(n \log n)$.
- Algoritmo de Kirkpatrick–Seidel con $O(n \log h)$.
- Algoritmo de Chan con $O(n \log h)$.

Donde n es le número de puntos y h el número de puntos que forman la envolvente convexa.

II. Metodología

En este aparatado se habla sobre consideraciones generales, y una breve descripción del programa implementado.

Solo se consideró el caso de 2 dimensiones y que el conjunto de puntos no tuviera duplicados, por lo que se leen de un archivo con n puntos (x, y) positivos, generado con un programa de *Mathematica*. En el caso de puntos colineales, solo se incluyen los de los extremos, o los más lejanos.

Los puntos que conforman la envolvente, son impresos en un archivo, para ser graficados en *Mathematica*.

Además de ser puntos positivos, se tiene un rango limitado de hasta 30 000, debido a un problema con el tipo de variable `int`, este asunto será explicado a más detalle en la siguiente sección.

Los algoritmos se basan en hallar la dirección relativa de 2 puntos, es decir, cuál está a la izquierda, y en el caso del Graham Scan, se ordenan según el ángulo polar. Aunque es posible calcular el ángulo directamente, esto es computacionalmente costoso, y realmente no es importante el valor del ángulo, solo el relativo; por lo que es posible usar el producto cruz que solo involucra restas y multiplicaciones, donde el signo indica la dirección o ángulo relativo.

II-A. Pseudocódigos

A continuación se presentan los pseudocódigos de los algoritmos en su forma más básica, sin lidiar con casos especiales.

II-A1. Jarvis March

En el Código 1 se presenta un pseudocódigo básico del algoritmo de Jarvis March. En este, se halla el punto más a la izquierda p_0, este es el punto de partida p. Luego, se considera el siguiente punto q en la lista, se escanean los puntos restantes r para checar si no hay ninguno a la izquierda de la línea formada por p y q, si es así, el punto r pasará a ser el nuevo punto q. Al terminar de escanear los puntos, se agrega al stack de resultados el punto p. Si se encuentra que el punto q es p_0, significa que ya se terminó el proceso.

```
JarvisMarch(puntos){
    n = length(puntos)
    p0 = índice del punto más a la izquierda
```

```
    p = p0

    while(true){
        q = (p+1)%n
        for(r = 0:n){

            ccw = pcruz(puntos[p], puntos[q],
            ↪  puntos[r])
            if(ccw > 0) //giro a la izquierda
                q = r
        }

        push(stack, puntos[p])

        if( q == p0)
            break;

        p = q
    }
}
```

Código 1: Pseudocódigo de algoritmo de Jarvis March.

II-A2. Graham Scan

En el Código 2 se presenta un pseudocódigo básico del algoritmo de Graham Scan. En este, se halla el punto inferior izquierdo p_0, el cual será la referencia para los otros, este se cambia a la posición 0 del arreglo de puntos; los puntos restantes se ordenan según su ángulo polar con referencia a p_0. Los primeros 3 puntos se ponen en el stack de resultados.

Se consideran los últimos dos puntos del stack y el siguiente en la lista r, mientras no se encuentre un giro a la izquierda, se retira un elemento del stack. Si se encuentra que hizo un giro a la izquierda, se agrega el punto r.

Se usó `qsort` para ordenar los puntos, usando como función de comparación el producto cruz, negativo, para que se ordenaran de forma descendente.

```
GrahamScan(puntos){
    n = length(puntos)
    ymin = índice del punto inferior izquierdo
    intercambiar(puntos[0], puntos[ymin])

    ordenar(puntos[1:end], ángulo polar)

    push(stack, puntos[0:2]) //primeros 3 puntos en
    ↪  el stack

    for(i = 3:n){

        while(true){
            ccw = pcruz(next(stack), top(stack),
            ↪  puntos[i])

            if(ccw > 0) //giro a la izquierda
                break;
            else
                pop(stack);
        }
        push(stack, puntos[i])
    }
}
```

Código 2: Pseudocódigo de algoritmo de Graham Scan.

II-A3. Algoritmo de Chan

En el Código 3 se presenta un pseudocódigo básico del algoritmo de Chan. Este combina los algoritmos anteriores. Hace k grupos con a lo mucho m puntos, y resuelve las k envolturas convexas usando Graham scan. Los puntos de las k envolturas, son la entrada para Jarvis March y calcular la envoltura final.

Para evitar crear copias de las secciones del arreglo correspondientes a cada grupo, solo se apunta a la dirección en la que esta presente el inicio y la longitud. Pero, los resultados de estos sí tuvieron que ser copiados a un arreglo adicional; aun así, son considerablemente menos que los de partida.

```
AlgoritmoChan(chan, puntos, m){
    n = length(puntos)
    k = n/m

    for(i = 0:k){//crear mini sets y calcular los
        chan->arreglos_chull[i]->puntos =
        ↪  &puntos[i*m] //para evitar hacer copias
        GrahamScan(chan->arreglos_chull[i]->puntos)
    }

    for(i = 0:k){
        //copiar resultados de cada uno a un nuevo
        ↪  set
        agregar(chan->puntos_mini,
        ↪  chan->arreglos_chull[i]->puntos_chull)
    }

    JarvisMarch(chan->puntos_mini)

}
```

Código 3: Pseudocódigo del algoritmo de Chan.

II-B. Estructuras empleadas

En el Código 4 se muestran las estructuras implementadas.

```
typedef struct {
    int x,y; //par de puntos x,y
} point;
```

```c
typedef struct {
    point *ptarray;
    size_t ssize;
} ptset;

typedef struct {
    point *ptarray;
    size_t ssize;
    int last_i; //indice de ultimo
} stack;

typedef struct {
    point start;
    size_t start_index;
    ptset *point_set; // el de puntos
    stack *ch_stk; //stack
    int h;
} convex_hull;

typedef struct{
    ptset *point_set;
    convex_hull **chull_array;
    int k; //k grupos
    int m; //con m puntos max
    ptset *points_minis;
    convex_hull *ch_minis; //para guardar los
    ↪  resultantes de los minis
} chan_chull;
```

Código 4: Estructuras implementadas.

II-C. *Funciones implementadas*

En el Código 5 se muestran los prototipos de funciones creados.

```c
ptset *new_randset(size_t set_size);
ptset *new_fileset(size_t set_size, char
↪  *file_name);
stack *new_stack(int n);
void stack_push(stack *stk, point pt);
void stack_pop(stack *stk);
point stack_next(stack *stk);
point stack_top(stack *stk);
int cross_product(point a, point b, point c);
int compare_angle(const void *gq, const void *gr);
convex_hull *new_convexhull(ptset *point_set);
convex_hull *new_convexhull_from_array(point *parr,
↪  int n_slice);
void chull_minx(convex_hull *chull);
void chull_minyx(ptset *point_set);
long long int dist_sq(point p, point q);
int compare_dist(point p, point q, point r);
void J_find_convexhull(convex_hull *chull);
void G_find_convexhull(convex_hull *chull);
void free_stack(stack *stk);
void free_ptset(ptset *pts);
void free_chull(convex_hull *chull);
void printFile_original(convex_hull *chull, char
↪  *fname);
void printFile_chull(convex_hull *chull, char
↪  *fname);
int compute_groups(int n, int m);
chan_chull *new_chan_chull(int k, int m, ptset
↪  *pts);
void C_find_convexhull(chan_chull *chan, char
↪  print);
void find_convex_hull(ptset *pts, char method, char
↪  print, int n, int m);
```

Código 5: Funciones implementadas.

II-D. *Código principal*

En el Código 6se presenta el uso del código, en donde, solo se tiene que crear una instancia del conjunto de puntos con la estructura `ptset`, leyendo n pares de un archivo con los puntos delimitados por `\t`, como ya se mencionó, este archivo es creado en *Mathematica*. También es posible generar estos puntos de manera aleatoria, pero no se puede asegurar el que no haya duplicados.

Con la función `find_convex_hull`, indicando el conjunto de puntos (creado anteriormente), el tipo de método: `jarvis`, `graham`, `chan`, si se quiere imprimir las salidas en archivos o no: `p`, `np` respectivamente, el número de puntos n y, para el caso del algoritmo de Chan, se debe de indicar el número máximo de puntos m para formar k grupos. Es recomendable que este m sea un divisor de n y mayor a 3.

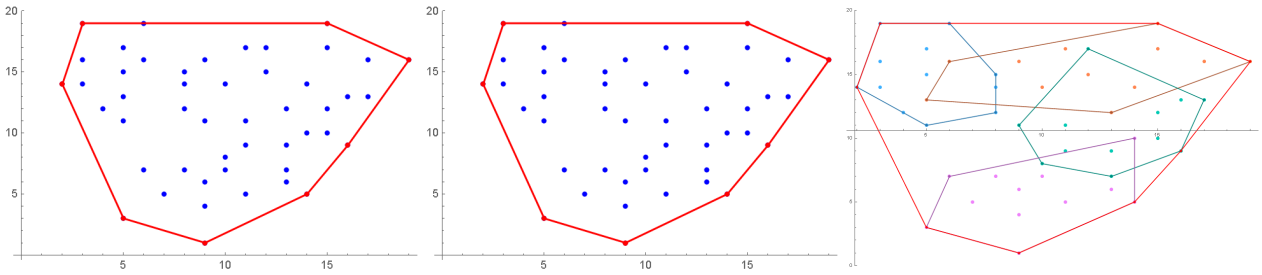
Los nombres de los archivos de salida tienen el nombre de `puntos_chull_jarvis.txt`, `puntos_chull_graham.txt` y `puntos_chull_chan.txt`. Para este último método, también se generan archivos correspondientes a cada grupo, tanto los puntos de partida como los de salida, con los nombres de archivo `puntos_originales_i.txt` y `puntos_chull_i.txt`, donde $0 \leq i < k$.

```c
int main(int argc, const char * argv[]) {

   int n =1000;
   int m = n/10;

    ptset *point_set = new_fileset(n,
    ↪  "./puntos_mexico_1k.txt");
    find_convex_hull(point_set, chan, p, n, m);

   return 0;
}
```

Código 6: Código principal.

II-E. *Funciones en Mathematica*

Las funciones incluyen la creación de archivos sin puntos duplicados, a partir de una lista, o según la geometría del territorio de algún país. También, la visualización de los archivos de salida del algoritmo, y también una especial



(a) Jarvis March (b) Graham Scan (c) Algoritmo de Chan

Figura 2: Visualización de resultados de los diferentes algoritmos.

para el Algoritmo de Chan para poder también visualizar las mini envolventes. En esta función solo se incluye el inicio del nombre de los archivos, indicando el número de archivos de salida (k grupos) y se enumeran automáticamente.

III. Resultados y discusión

En la Figura 2 se muestra un ejemplo de la visualización de la salida de los 3 métodos y en la Figura 3 se presenta una comparación de los tres algoritmos: tiempo de ejecución (s) vs. número de puntos. Es posible que el algoritmo de Chan no se haya desempeñado "tan bien"debido a la selección de m, ya que este también es un hiperparámetro. Aun así, a pesar de tener envuelto varios ciclos, su desempeño fue considerable.

Dado que los puntos eran relativamente aleatorios, el número de puntos de la envolvente convexa eran menos, por lo que se esperaba que un algoritmo como el de Jarvis March, con sensibilidad a la salida, se desempeñara bien.

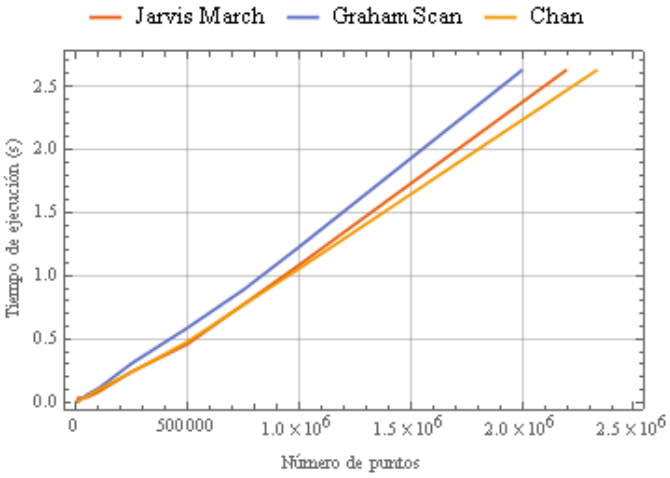


Figura 3: Comparación de tiempo de ejecución - número de puntos.

III-A. Errores y limitaciones

Hay problemas para liberar la memoria con `free`, a veces causa que el proceso termine erróneamente; no se encontró la causa de esto.

Otro error sin resolver aún, es que al leer archivos con números grandes que requieren de `long long int`, el proceso no termina bien o no lee correctamente el número de puntos si es cercano a potencias de 2 ($2^x - 1$). A pesar del uso de `%lld` en `sscanf` este no lee correctamente los puntos, además de que causa advertencias por el compilador. Se encontró que, `%lld` y `%I64d` no es soportado por todos los compiladores. Es por esto que está limitado a valores de hasta aproximadamente 60k.

Falta ser adaptado para que este pueda ser ejecutado a través de la terminal; además de poner los archivos de salida en un lugar más apropiado (carpetas independientes), ya que dependen del directorio de trabajo actual.

IV. Conclusión

El código no es lo suficientemente robusto, ya que no tiene suficiente manejo de errores, por lo que si no se cumplen las suposiciones hechas, es posible que el proceso termine de forma inesperada o que no se obtengan los resultados deseados. Además, de que depende del directorio de trabajo actual, sobre todo para colocar los archivos de salida, y leer el de entrada; y no ha sido adaptado para poder ser ejecutado en la terminal.

Sin embargo, satisfaciendo las condiciones ya mencionadas, los algoritmos implementados se desempeñan bien, y pueden encontrar el convex hull en cuestión de segundos. El algoritmo de Chan podría ser paralelizado, de manera un tanto más directa para mejorar su desempeño.

Ha de considerarse durante la etapa de diseño, el formato o tipo de variables a emplear y si habrá operaciones que hagan que se supere el límite soportados por el tipo de variable (por ejemplo, multiplicaciones), además de la precisión de los resultados de estas operaciones. De igual manera, se debe de buscar diferentes rutas para obtener resultados similares con el uso de operaciones menos complejas; en este caso, usar producto cruz que implica solo restas y multiplicaciones, a comparación de calcular el ángulo polar que implica operaciones trigonométricas y divisones.