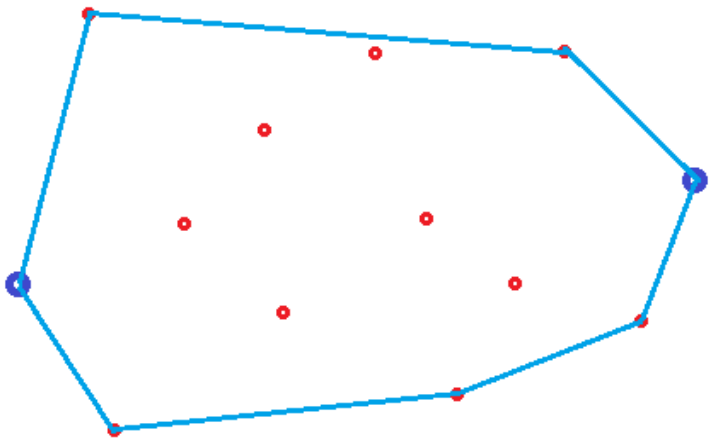
75.29 - Teoría de Algoritmos

TP Nº 2

93542 - Bollero, Carlos
98059 - Czop, Guillermo
98017 - Errázquin, Martín
85826 - Escobar, Cynthia

1 El robot y el camino seguro

Un robot debe llegar desde el punto en el que se encuentra a un destino determinado. Disponemos de un mapa que contiene un conjunto de puntos seguros (El punto inicial y el final son puntos seguros). Sabemos que existen accidentes geográficos donde el robot puede quedar atrapado. Estos accidentes se encuentran dentro de la envoltura convexa que contienen a los puntos seguros (aclaración: los puntos de inicio y finalizacion pertenecen a la envoltura convexa).



El objetivo final es, dados un input que contiene los puntos de origen, final y seguros, determinar el menor camino seguro posible. Para eso calcular la envoltura convexa. Los dos caminos posibles utilizando como ruta la envoltura. La selección del menor camino utilizando la distancia euclídea.

Se solicita:

Resolver el problema utilizando diferentes algoritmos para calcular la envoltura convexa.

- Fuerza bruta
- Graham scan
- Un método de división y conquista.

Para cada algoritmo se solicita:

1. Explicar el método, presentar el pseudocódigo y las estructuras a utilizar.
2. Calcular y explicar la complejidad de su algoritmo (tiempo y espacio)
3. Programar la solución.
4. Explique y analice la reducción polinomial realizada para el problema planteado. Podemos afirmar que nuestro problema es P (desarrolle)?

Información adicional:

Utilizar como entrada un archivo de texto con el siguiente formato:

- Por cada línea dos números representando las coordenadas X e Y de puntos seguros separados por un espacio (números enteros).
- La primera línea corresponde al punto de origen.
- La segunda línea corresponde al punto de destino.

La salida debe ser mostrada en pantalla con el siguiente formato:

```
Camino 1: Longitud [poner longitud del camino]
Recorrido: [mostrar el x,y de cada punto iniciando desde origen y pasando en forma ordenada por los pun

Camino 2: Longitud [poner longitud del camino]
Recorrido: [mostrar el x,y de cada punto iniciando desde origen y pasando en forma ordenada por los pun

Camino seleccionado: [1 o 2]
```

El programa debe recibir por parámetro el nombre del archivo con los puntos y el tipo de algoritmo a utilizar (F: Fuerza bruta, G: Graham scan, D: División y conquista)

Fuerza bruta

1.
   - Detalle

     Dado el conjunto de puntos seguros $P = \{p_1, p_2, .., p_n\}$, para cada par de puntos p_i, p_j, $i \neq j$, construimos el segmento de recta $s_{ij} = (p_i, p_j)$.
     El segmento $s_{ij} = (p_i, p_j)$ formará parte de la envoltura convexa si el resto de los puntos p_k con $k \neq i$ y $k \neq j$, se encuentran todos en un lado del segmento.

   - Pseudocódigo

```
calcular_envoltura_convexa(p):
    envoltura_convexa=[]
    para i desde 0 hasta n:
        para j desde 0 hasta n:
            si i == j:
                continue

            todos_del_mismo_lado = true

            para k desde 0 hasta n:
                si k == i || k == j:
                    continue

                si orientacion(p[i],p[j],p[k]) < 0:
                   todos_del_mismo_lado = false
                   break
```

```
            si todos_del_mismo_lado:
                envoltura_convexa.add(p[i],p[j])

    return envoltura_convexa
```

Sean $A = p[i]$, $B = p[j]$ y $C = p[k]$, la función **orientacion** es el determinante de los vectores $\overrightarrow{AB}$ y $\overrightarrow{AC}$:

$$det(\overrightarrow{AB}, \overrightarrow{AC}) = (B_x - A_x)(C_y - A_y) - (B_y - A_y)(C_x - A_x)$$

y determina de que lado del segmento de recta $\overline{AB}$ se encuentra el punto C. Vale notar que la orientación del punto C varía si se "mira" desde A hacia B o de B a A, y más precisamente son opuestas.
Dado que se probarán todas las combinaciones de puntos posibles, se evaluarán tanto el segmento $\overline{AB}$ como $\overline{BA}$, así que nos alcanza con verificar que la orientación de todos los puntos k con $k \neq i, j$ sea, por ejemplo, > 0 para confirmar que se encuentran de un mismo lado de $\overline{AB}$.

- Estructuras

  Lista de puntos seguros y envoltura convexa

2. Complejidad

   - La complejidad es $O(n^3)$ en tiempo ya que debemos armar un segmento de recta para cada par de puntos p_i, p_j, con lo que a cada punto $p_i, 0 \leq i \leq n$, lo tenemos que combinar con cada punto $p_j, 0 \leq j \leq n$, es decir $O(n^2)$. Luego al par (p_i, p_j) deberemos compararlo con los $n - 2$ puntos restantes para determinar si realmente el segmento pertenece a la envoltura convexa.
   - $O(n)$ en espacio ya que sólo necesitamos mantener una lista con los n puntos seguros y la lista con los puntos pertenecientes al mínimo camino seguro.
     NOTA: Revisando el código se encontró que estabamos empeorando la complejidad espacial ya que se utilizaba una estructura auxiliar (un diccionario de listas) para guardar las asociaciones formadas por las coordenadas y determinar si los puntos S y T eran incluidos en el camino encontrado. Para esta reentrega se la quitó, ya que era innecesaria.

3. Ver apéndice.

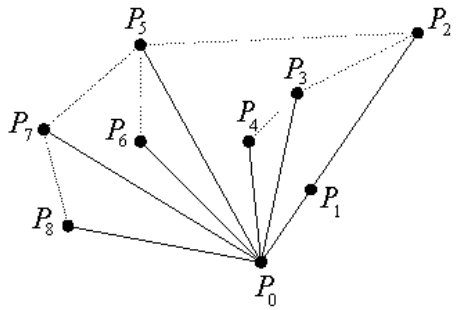
Graham scan

1. • Detalle

   Dado el conjunto de puntos seguros $P_n = \{p_1, p_2, .., p_n\}$, donde cada punto p_i es una coordenada en el plano (p_{i_x}, p_{i_y}). Este algoritmo consiste en:

   ◦ Encontrar el punto p_i con la menor componente p_{i_y}, en caso de empate nos quedaremos con el punto con menor componente p_{i_x}. Llamaremos p_0 a este punto.

   ◦ Calcular para cada punto p_i el ángulo formado entre el segmento que los une con el punto p_0 y el eje de abscisas, y ordenarlos crecientemente.



   ◦ Del conjunto de puntos, remover los que formen un ángulo cóncavo con los dos anteriores: recorrerlos manteniendo referencia a la terna compuesta por los 3 últimos puntos $t_i = (p_{i-1}, p_i, p_{i+1}), 2 \leq i$ y chequear que la orientación del ángulo formado por ellos no tenga sentido horario.

   ◦ Finalmente, se completa el proceso al volver al punto de partida y los puntos que no fueron removidos forman la envolvente convexa.

• Pseudocódigo

```
envoltura_convexa(puntos):
    envoltura := []
    min := punto de menor coordenada ordenada
    puntos_ordenados := ordenar_por_angulo()
    envoltura.push(min)
    envoltura.push(puntos_ordenados[0])
    para i desde 1 hasta puntos_ordenados.size():
        mientras (angulo envoltura.top(), envoltura.nextToTop() y puntos_ordenados(i) <= 0):
            envoltura.pop()
        envoltura.push(puntos_ordenados(i))
    return envoltura
```

• Estructuras

   ◦ Lista de puntos ordenados

   ◦ Pila de puntos envoltura convexa

2. Complejidad

   Encontrar el punto inferior o mínimo tiene complejidad $O(n)$. El ordenamiento de los puntos tiene una complejidad $O(nlogn)$. Luego, en el peor de los casos, se procesa cada punto dos veces; una vez como nuevo punto a agregar a la envolvente en un giro a izquierda y otra vez como tope de la pila envolvente en un giro a derecha, en cuyo caso se elimina. La complejidad de iterar sobre los puntos ordenados es $O(n)$. La complejidad del proceso de ordenamiento prevalece por lo tanto la complejidad temporal del algoritmo Graham Scan es $O(nlogn)$.

   La complejidad espacial es $O(n)$, ya que sólo debemos mantener una lista con los puntos y la pila de puntos pertenecientes a la envolvente convexa.

3. Ver apéndice.

División y Conquista

1. • Detalle

Dado el conjunto de puntos seguros $P_n = \{p_1, p_2, .., p_n\}$, este método responde a la idea de que encontrar la envoltura convexa de k conjuntos de puntos P_k con $|P_k| < |P_n|$ tal que $\bigcup_{i=1}^{k} P_i = P_n$, es más fácil que encontrar la envoltura convexa de P_n.
La idea general de este algoritmo es ordenar los puntos por su componente x y recursivamente:

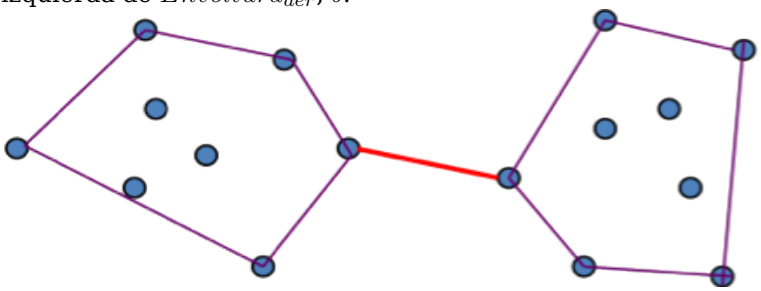
(a) Si la longitud de la entrada es menor o igual a 6, se procede a calcular la envoltura convexa por Fuerza Bruta.

(b) Si no, dividir en dos el conjunto de puntos de entrada, obteniendo un arreglo con los puntos más a la derecha (P_{der}) y otro con los puntos más a la izquierda (P_{izq}).

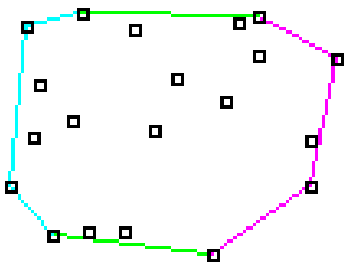
(c) Calcular volver al punto $a)$ para P_{der} y P_{izq}, obteniendo $Envoltura_{der}$ y $Envoltura_{izq}$.

(d) Finalmente se procede a mergear $Envoltura_{der}$ y $Envoltura_{izq}$ calculando los segmentos de linea que unirá a ambas envolturas, llamados tagentes superior y inferior. El cálculo de la tangente superior es el siguiente (la tangete inferior se calcula de manera simétrica):

- Se ordenan los puntos en sentido antihorario.
- Se toman inicialmente el punto más a la derecha de $Envoltura_{izq}$, a, y el punto más a la izquierda de $Envoltura_{der}$, b.



- Se valida que no haya ningún punto de $Envoltura_{izq}$ y $Envoltura_{der}$ por encima de $\overline{ab}$. Si un punto estuviera por encima significa que $\overline{ab}$ no es la tangente superior, y el escenario se resuelve:
  - si un punto de $Envoltura_{der}$ quedó por encima, entonces se mueve el punto b uno hacia arriba dentro de $Envoltura_{der}$.
  - si un punto de $Envoltura_{izq}$ quedó por encima, entonces se mueve el punto a uno hacia arriba dentro de $Envoltura_{izq}$.
- A continuación se observa en verde las tangetes superior e inferior que terminan de cerrar la envoltura final uniendo $Envoltura_{der}$ y $Envoltura_{izq}$.



- Pseudocódigo

```
mergear(derecha,izquierda):
    derecha = ordenar_por_sentido_antihorario(derecha)
    izquierda = ordenar_por_sentido_antihorario(izquierda)

    a=buscar_punto_mas_der(izquierda)
    b=buscar_punto_mas_izq(derecha)

    tangente_sup_encontrada = false

    mientras !tangente_sup_encontrada:
        mientras haya puntos en derecha por encima de ab
            b = siguiente punto hacia arriba en derecha
        mientras hayas puntos en izquierda por encima de ab
            a = siguiente punto hacia arriba en izquierda
        tangente_sup_encontrada = true

    tangente_superior = ab

    tangente_inf_encontrada = false

    mientras !tangente_inf_encontrada:
        mientras haya puntos en derecha por debajo de ab
            b = siguiente punto hacia abajo en derecha
        mientras hayas puntos en izquierda por debajo de ab
            a = siguiente punto hacia abajo en izquierda
        tangente_inf_encontrada = true

    tangente_inferior = ab

    envoltura = conectar derecha e izquierda con tangente_superior y tangente_inferior

    return envoltura

dividir(p):
    si len(p) <6:
        return fuerza_bruta(p)
    mitad = len(p)//2
    derecho = p[mitad:]
    izquierdo = p[:mitad]

    envolutura_der=dividir(derecho)
    envolutura_izq=dividir(izquierdo)

    return mergear(envoltura_der,envoltura_izq)

calcular_envoltura_convexa(p):
    envoltura_convexa=[]
    ordenar_por_x(p)
    return dividir(p)
```

- Estructuras
  - Lista de puntos ordenados
  - Lista de puntos envoltura convexa

2. Complejidad

   - $O(nlogn)$ en tiempo ya que el mergeo de las envolturas izquierda y derecha es $O(n)$, pero a su vez vamos diviendo las entradas en 2 arreglos del mismo tamaño, con lo que la complejidad final es $O(nlog(n))$
   - $O(n)$ en espacio ya que sólo necesitamos mantener una lista con los n puntos seguros y la lista con los puntos pertenecientes al mínimo camino seguro.

3. Ver apéndice.

Reducción

Dado $Y =$ Robots, es el problema Y polinomial?
Para determinar si Y pertenece a la clase P trataremos de reducirlo a un problema conocido que sabemos sí pertenece, por ejemplo X = Envoltura Convexa. Es decir que verificaremos si algún algoritmo que resuelve X, Envoltura Convexa, en tiempo polinomial, por ejemplo División y conquista en $O(n\ log(n))$, nos ayuda a resolver Y.
Probaremos entonces $Y \leq_p X$, para lo que definiremos:

1. Para X:

   - La entrada $X_{In} = \{x_1, x_2, .., x_n\}$, con $x_i, i \in [1, n]$, puntos en el plano.
   - La salida $X_{Out} = \{\{x_j, .., x_k\} / j \neq k \wedge j, k \in [1, n]\}$, la envoltura convexa de X, es decir, el mínimo subconjunto de puntos $\in X_{In}$ que forma una envoltura que contenga al resto de los puntos de X_{In}.

2. Para Y:

   - La entrada $Y_{In} = \{s, t, .., y_n\}$, donde $s = y_1$ es la posición actual del robot y $t = y_2$ es el destino al que quiere llegar, con $y_i, i \in [1, n]$, puntos en el plano.
   - La salida $Y_{Out} = \{s, .., t\} = \{y_1, .., y_k, .., y_2\}$, es el menor camino seguro en Y, es decir, el mínimo subconjunto de puntos $\in Y_{In}$ que permiten al robot moverse desde s a t sin quedar atrapado en algún accidente geográfico.

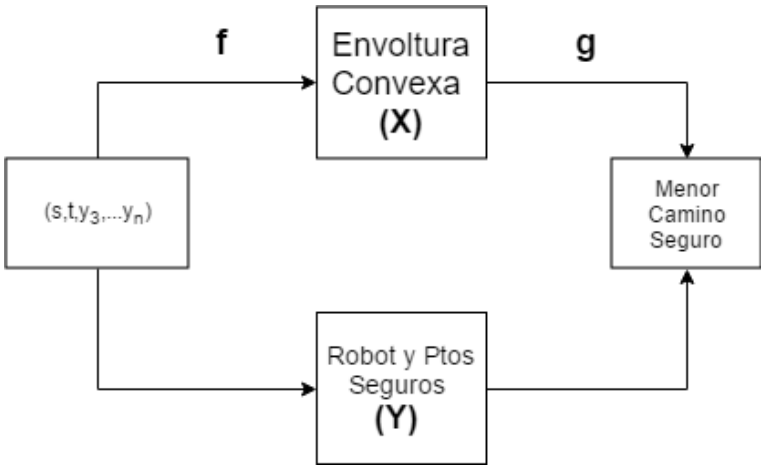
3. La función **f** / $\mathbf{f}(Y_{In}) -> X_{In}$, que en este caso en particular no realizará ninguna transformación, es decir que $\mathbf{f}(Y_{In}) = Y_{In} = X_{In}$.

4. La función **g** / $\mathbf{g}(X_{Out}) -> Y_{Out}$, deberá obtener de la envoltura X_{Out} los dos caminos que unen s con t. Pseudocódigo de la función **g**:

```
X_ordenado = ordenar_por_sentido_antihorario(X_Out)
pos_t = pos(X_ordenado, t)
pos_s = pos(X_ordenado, s)
si pos_t > pos_s
    camino_1 = X_ordenado[pos_s a pos_t]
    camino_2 = reverso(X_ordenado[pos_t a len(X_ordenado)] + X_ordenado[0 a pos_s])
sino
    camino_1 = X_ordenado[pos_s a len(X_ordenado)] + X_ordenado[0 a pos_t]
    camino_2 = reverso(X_ordenado(pos_t a pos_s))

si peso(camino_1) > peso(camino_2)
    return camino_2
sino
    returno camino_1
```

La complejidad de la función **g** es $O(nlog(n))$, ya que lo más costoso es ordenar el conjunto de puntos X_{Out}, es decir que tiene complejidad polinomial.



Por lo anterior, podemos decir que el problema Y pertenece a la clase P ya que la complejidad del algoritmo de división y conquista que soluciona X es polinomial y el costo de la reducción está dado unicamente por el costo de aplicar **g** a X_{Out} que vimos también tiene una complejidad polinomial.

2 Una variante de SAT

2SAT o (2-satisfiability) es un caso particular del problema SAT. En este caso las cláusulas lógicas comprenden únicamente 2 variables.

1. Definir una reducción de un problema de 2SAT a un problema de 3SAT
2. Investigar y responder justificando la validez de la siguiente afirmación: "Es posible resolver una instancia de 2SAT utilizando una reducción polinomial a un problema de grafos, a su vez resoluble en tiempo polinomial"
3. Analice la implicancia de los puntos 1 y 2 a la luz del problema de P=NP

2.1 Reducción de 2SAT a 3SAT

Nótese primero que *k*-SAT es el problema de satisfacibilidad para cláusulas de *hasta k* elementos. Entonces, una CNF de 2-SAT es también un caso particular de 3-SAT y, en general, de cualquier *k*-SAT con $k > 2$. Luego, resulta trivial reducirlo porque suponiendo un solucionador de 3-SAT, una solución de 2-SAT es resolverlo como caso particular de 3-SAT.
Considerando X una 2-CNF, solve_3_SAT: 3-CNF $\rightarrow \{0,1\}^n$ un solucionador de 3-SAT,

```
solve_2_SAT(X):
    return solve_3_SAT(X)
```

2.2 Resolución polinomial de 2-SAT

Para mostrar que es verdad vamos a enunciar primero algunos puntos que nos serán de utilidad:

1. Una cláusula de 1 elemento es equivalente a una OR del elemento consigo mismo. Por ejemplo, $x \equiv x \vee x$.
2. Una OR lógica de 2 elementos es equivalente a una AND de las implicaciones de uno negado al otro. Por ejemplo, $x \vee y \equiv (\overline{x} \rightarrow y) \wedge (\overline{y} \rightarrow x)$

De (1) y (2) vemos que cualquier 2-CNF de *c* cláusulas se puede expresar como una AND de *2c* implicaciones. En adelante vamos a utilizar esta expresión, que además es obtenible en tiempo lineal porque es desdoblar cada cláusula.

Consideramos G = (V,E) el digrafo cuyos vértices son las variables (negadas y no) de la 2-CNF, y que $\forall u, v \in V : (u, v) \in E \Leftrightarrow (u \Rightarrow v)$ es una cláusula de la 2-CNF. Obtuvimos entonces (también en tiempo lineal) el llamado "grafo de implicaciones" G.
Sobre G corremos el algoritmo de Tarjan para obtener las componentes fuertemente conexas (CFC) y su orden topológico inverso (OTI).
Para cada componente conexa verificamos que ninguna variable esté al mismo tiempo que su complemento. Si se detecta algún caso, la expresión booleana no es satisfacible.
Siguiendo el OTI obtenido de Tarjan, para cada CFC asignar 1 a todas sus variables que no tuvieran ya valor (y consecuentemente 0 a sus complementos). Al finalizar, se ha obtenido una asignación para las variables que satisface la expresión.

Obsérvese que para todo el grafo trabajamos en O(|V|+|E|) y que construimos el grafo con $|V| = 2n, |E| = 2c$ por lo que todo el algoritmo es O(n+c).

2.3 Implicancia sobre P vs NP

Sabemos que:

- 3SAT $\in$ *NP-C*
- De 2.1, 2-SAT $\leq_P$ 3-SAT
- De 2.2, 2-SAT $\in$ *P*

Notemos que si P $\neq$ NP, entonces 2-SAT $<_P$ 3-SAT. Ahora, si P = NP, entonces P = NP = NP-C y por tanto 2-SAT $=_P$ 3-SAT. En sentido inverso, veamos que en un principio podría haberse pensado que 2-SAT era NPC como el resto de la familia de k-SAT, sin embargo en 2.2 vimos que hay al menos una forma conocida de resolverlo en tiempo polinomial, lo cual nos deja el interrogante de si no hay un correspondiente para 3-SAT. Si lo hubiera, se estaría probando que un problema NPC (y por tanto todos los demás) está en P, y se habría probado que P = NP.

3 El problema de los pasantes

Un grupo de pasantes de un museo de antropología se han metido en un problema. Sin querer han tirado algunos cajas y mezclado colecciones de X objetos recolectados en dos excavaciones. Dado su inexperiencia, su apuro y la similitud de las piezas tienen como idea comparar de a pares para intentar volver a separarlos. Por cada par de toman definen si pertenece a la misma colección, diferente o si no están seguros.

1. Ayude a los pasantes a determinar mediante un algoritmo eficiente si es posible separar las colecciones basándose en sus clasificaciones. (ayuda: pruebe una solución utilizando grafos)
2. Determine el orden de complejidad del algoritmo propuesto.
3. Ejemplifique el funcionamiento de su algoritmo con un caso donde se pueda clasificar y otro en el que no.

3.1 Solución propuesta

El objetivo del algoritmo planteado es encontrar una 2-coloración del grafo formado por los vértices (que representan los objetos de las colecciones) y las aristas (que representan las comparaciones realizadas por los pasantes). El grafo a utilizar no contiene las aristas correspondientes a comparaciones donde los pasantes no estén seguros si los items pertenecen a la misma colección o no ya que no aportan información al problema.
Hallar una 2-coloración equivale a analizar si el grafo es bipartito, donde cada color color equivale a una parte.
Aquellas aristas que unen 2 elementos de una misma colección pertenecen al subconjunto de aristas llamado "E_SI". Mientras que las aristas que unen elementos de distintas colecciones pertenecen al subconjunto denominado "E_NO".
Por lo tanto se parte del grafo G=(V, E_SI+E_NO).
Se deben cumplir 2 condiciones para que los elementos puedan categorizarse:

1. El grafo debe tener una única componente conexa.
2. Debe haber una 2-coloración del grafo donde cada par de vértices unidos por una arista "E_SI" posean el mismo color y cada par de vértices unidos por una arista "E_NO" posean colores distintos.

La primer condición se puede verificar fácilmente utilizando tanto el algoritmo de DFS como BFS. Partiendo de un vértice cualquiera se deben visitar todos los demás vértices del grafo. En caso de quedar al menos uno sin visitar se sabe que se tienen 2 o más componentes conexas distintas y no se puede agrupar los elementos en las 2 colecciones.
Suponiendo un grafo conexo, la segunda condición se puede analizar utilizando BFS modificado. Partiendo de un vértice cualquiera con un color ya asignado (no importa cuál sea) se van visitando los demás vértices y pintándolos según la regla:

1. Si el vértice no visitado está unido por una arista "E_SI" se lo pinta del mismo color que el vértice origen.
2. Si el vértice no visitado está unido por una arista "E_NO" se lo pinta del color opuesto que el vértice origen.

Cuando se procesa un vértice que posea un vecino ya coloreado previamente se compara el color asignado a dicho vértice con el color esperado siguiendo la regla mencionada previamente. En caso de que no coincidan significa que hay una contradicción y los elementos no pueden ser agrupados en las 2 colecciones.
A continuación se presenta el pseudocódigo del BFS modificado:

```
clasificacion(u):
        queue.add(u)
        while (!queue.empty()):
                v = queue.get()
                if visitado[v] == false:
                        for each r in v.adyacentes():
                                if (visitado[r] == false):
                                        queue.add(r)
                                if colour(r) == null:
                                        if arista(v,r) in E_SI:
                                                colour(r) = colour(v)
                                        else:
                                                colour(r) = opposite(colour(v))
                                else:
```

```
                                        if expected_colour(color(v),arista(v,r)) != colour(r):
                                            return false
                        visitado[v] = true
        return true
```

Los colores de los vértices se almacenan en un diccionario donde el vértice mismo es la llave y que posee como valor el color del mismo. Así mismo se utiliza un diccionario cuyas llaves son de la forma (v1,v2), vértices del gráfo, y el valor indica si la arista que une dichos vértices pertenece a "E_SI" o a "E_NO".

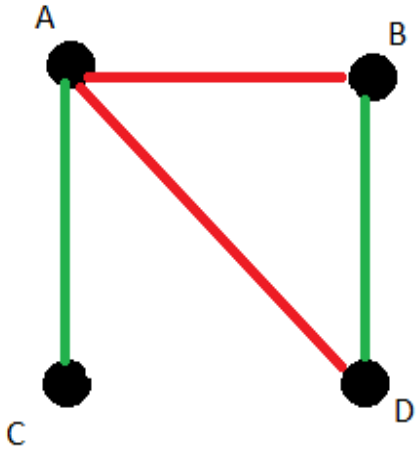
3.2 Complejidad algorítimica

La solución planteada tiene como complejidad $O(|V| + |E\_SI| + |E\_NO|)$ dado que solo se utiliza BFS (normal y modificado) para analizar las condiciones requeridas. El algoritmo modificado tiene la misma complejidad que el original ya que no se realizan operaciones costosas gracias a que se utilizan diccionarios para acceder rápidamente a la información de los colores y aristas ($O(1)$). La inicializacion del diccionario de aristas es de órden $O(|E\_SI| + |E\_NO|)$, que es menor que la complejidad de BFS.

3.3 Ejemplos

Para ambos casos se marcan en verde las aristas que unen elementos de una misma colección (aristas pertenecientes a E_SI) mientras que las aristas rojas unen elementos de colecciones distintas (aristas E_NO).

Caso exitoso



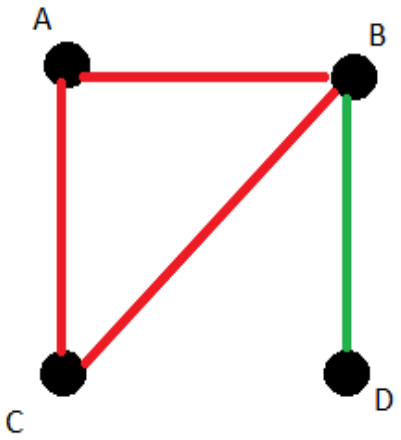
Se puede ver a simple vista que el grafo posee una única componente conexa, cumpliendo así la primera condición explicada.

Para comprobar la segunda condición se utilizará el algoritmo previo partiendo desde el vértice A y asignandole el color "C1" (siendo "C2" el color restante).

A posee 3 vecinos sin visitar ni pintar: B,D,C. El vértice B es pintado con el color "C2" ya que se conecta por una arista "E_NO" y luego se lo suma a la cola ([B]). Al vértice D le sucede lo mismo, quedando marcado con el color "C2" y agregado a la cola ([B,D]). Finalmente como el vértice C está unido a A mediante una arista "E_SI" se lo pinta con el mismo color de A, es decir "C1". Al finalizar esto la cola queda con los 3 elementos [B,D,C].

Se desencola el vértice B. Como tanto D y A ya fueron pintados solo se debe verificar que los colores sean los correctos. La arista BD pertenece al subconjunto "E_SI" por lo que el color esperado es el mismo de B. Se cumple que ambos vértices hayan sido marcados con el mismo color "C2". La arista BA es del tipo "E_NO" por lo que debe tener el color opuesto, "C1". Nuevamente esto se cumple.
Se continua el proceso para los vértices restantes y se verifica que no hay ningún caso donde una arista "E_SI" une vértices de colores distintos ni una arista "E_NO" une vértices del mismo color. Al terminar queda la 2-coloración C1={A,C} y C2={B,D}, quedando así 2 colecciones separadas.

Caso fallido



Se puede ver a simple vista que el grafo posee una única componente conexa, cumpliendo así la primera condición.
Se parte del vértice A (color "C1"). Como está unido a B y C mediante aristas del tipo "E_NO" se los pinta a ambos con el color "C2" y se los encola ([B,C]).
Se pasa a procesar al vértice B (cola: [C]). Este tiene como vecinos a D y C. Como D no posee ningún color y la arista BD pertenece a "E_SI" D es coloreado con "C2" y se lo suma a la cola ([C,D]). Como C ya tiene un color asignado se debe comprobar que posea el color esperado. Como B fue pintado con "C2" y la arista BC está dentro de las "E_NO" se espera que C haya sido pintado con el color "C1". Pero esto no se cumple. C tambíen fue pintado con el color "C2" por lo que hay una inconsistencia en el grafo. Esto significa que no se puede realizar una 2-coloración del grafo, no cumpliendo así la segunda condición necesaria.
Como solamente cumple la primera condición de éxito, los vértices del grafo no pueden ser agrupados en 2 colecciones distintas.

4 Requisitos y ejecución

- Requisitos:
  Python3
- Ejecución:
  - ./safepath.py inputFile F|G|D

- Apéndice - Fuerza Bruta & Division y Conquista

```
#!/usr/bin/env python

import sys
import time
import math
import os
import operator
import functools
from collections import deque, namedtuple
import grahamScan

Edge = namedtuple('Edge', 'start, end, distance')
Coordinate = namedtuple('Coordinate', 'x,y')
WeightedPath = namedtuple('WeightedPath', 'path,weight')

centroid = Coordinate(0,0)

#####
def get_key(item):
    return item.x

def distance(p1, p2):
        return math.sqrt(pow(p1.x-p2.x, 2) + pow(p1.y-p2.y, 2))

## Checks whether the line is crossing the polygon,
# Se utiliza para saber de que lado esta el punto c con respecto
# a la linea segmento de linea ab
def orientation(a, b, c):
        res = (b.y-a.y)*(c.x-a.x) - (c.y-a.y)*(b.x-a.x)

        if res == 0:
                return 0
        if res > 0:
                return 1
        return -1

def quad(p):
        if (p.x >= 0 and p.y >= 0):
                return 1
        if (p.x <= 0 and p.y >= 0):
                return 2
        if (p.x <= 0 and p.y <= 0):
                return 3
        return 4
```

```
def counterclockwise_sorting(item1, item2):
        result = 0

        p = Coordinate(x=(item1.x-centroid.x),y=(item1.y-centroid.y))
        q = Coordinate(x=(item2.x-centroid.x),y=(item2.y-centroid.y))

        quad_item1=quad(p)
        quad_item2=quad(q)

        if quad_item1 != quad_item2:
                result = -1 if quad_item1 < quad_item2 else 1
        else:
                result = -1 if (p.y*q.x < q.y*p.x) else 1

        return result

def merger(left_hull,right_hull):

        len_left = len(left_hull)
        len_right = len(right_hull)

        idx_righmost_a = 0
        idx_leftmost_b = 0

        for i in range(1,len_left):
                if left_hull[i].x > left_hull[idx_righmost_a].x:
                        idx_righmost_a=i

        for i in range(1,len_right):
                if right_hull[i].x < right_hull[idx_leftmost_b].x:
                        idx_leftmost_b=i

        #finding the upper tangent
        inda = idx_righmost_a
        indb = idx_leftmost_b
        done = False
        while not done:
                done = True

                while (orientation(right_hull[indb], left_hull[inda],
                            left_hull[(inda+1)%len_left]) > 0):
                        inda = (inda+1)%len_left

                while (orientation(left_hull[inda],right_hull[indb],
                        right_hull[(len_right+indb-1)%len_right]) < 0):
                        indb = (len_right+indb-1)%len_right
                        done = False

        uppera = inda
        upperb = indb
        inda = idx_righmost_a
        indb = idx_leftmost_b
        done = 0
```

```
# The edge ab is a tangent if the two points
        about a and the two points about b are on the same side of ab.

g = 0
# finding the lower tangent
while not done:
        done = 1
        while (orientation(left_hull[inda], right_hull[indb],
                    right_hull[(indb+1)%len_right])>0):
                indb=(indb+1)%len_right

        while orientation(right_hull[indb], left_hull[inda],
                    left_hull[(len_left+inda-1)%len_left])<0:
                inda=(len_left+inda-1)%len_left
                done=0

lowera = inda
lowerb = indb
ret = []

# ret contains the convex hull after merging the two convex hulls
# with the points sorted in anti-clockwise order
ind = uppera

last_point = left_hull[uppera]
ret.append(left_hull[uppera])

while (ind != lowera):
        ind = (ind+1)%len_left
        new_point = left_hull[ind]
        ret.append(new_point)
        last_point = new_point

ind = lowerb

new_point = right_hull[lowerb]

last_point = new_point

ret.append(right_hull[lowerb])

while ind != upperb:
        ind = (ind+1)%len_right
        new_point = right_hull[ind]
        ret.append(new_point)
        last_point = new_point

return ret
```

```
def divide(s,t,points):
        if len(points) > 1 and len(points) <= 6:
                return fuerza_bruta(s,t,points)

        half = len(safe_points)//2
        right=points[half:]
        left=points[:half]

        fuerza_bruta_der = divide(s,t,right)
        fuerza_bruta_izq = divide(s,t,left)

        if (s not in fuerza_bruta_der and s not in fuerza_bruta_izq):
                print("ERROR: Convex Hull does not contain source.")
                sys.exit(500)

        if (t not in fuerza_bruta_izq and t not in fuerza_bruta_der):
                print("ERROR: Convex Hull does not contain tail.")
                sys.exit(500)

        merge = merger(fuerza_bruta_izq, fuerza_bruta_der)

        return merge
#####

def fuerza_bruta(s, t, safe_points):
        global centroid
        minimum_convex_hull = []

        for i in range(0, len(safe_points)):
                for j in range(0, len(safe_points)):
                        if i == j:
                                continue

                        pointI = safe_points[i]
                        pointJ = safe_points[j]

                        allPointsOnTheRight = True

                        for k in range(0, len(safe_points)):
                                if k == i or k == j:
                                        continue

                                d = orientation(pointI, pointJ, safe_points[k])
                                if d < 0:
                                        allPointsOnTheRight = False
                                        break

                        if allPointsOnTheRight:
                                if pointI not in minimum_convex_hull:
                                        minimum_convex_hull.append(pointI)
                                if pointJ not in minimum_convex_hull:
                                        minimum_convex_hull.append(pointJ)
```

```
        x = [p.x for p in minimum_convex_hull]
        y = [p.y for p in minimum_convex_hull]

        centroid = Coordinate(x=(sum(x) / len(minimum_convex_hull)), y=(sum(y) /
                    len(minimum_convex_hull)))

        sorted_minimum_convex_hull = sorted(minimum_convex_hull,
                key=functools.cmp_to_key(counterclockwise_sorting))

        return sorted_minimum_convex_hull

def armar_caminos(s,t,convex_hull):
        idx_s = convex_hull.index(s)
        idx_t = convex_hull.index(t)

        weight_1 = 0
        weight_2 = 0

        if idx_t < idx_s:
                path1 = convex_hull[idx_s:]
                path1.extend(convex_hull[0:idx_t+1])
        else:
                path1 = convex_hull[idx_s:idx_t+1]

        path2 = []
        path2.append(s)

        if idx_t > idx_s:
                if idx_s != 0:
                        path2.extend(reversed(convex_hull[:idx_s]))

                if idx_t != len(convex_hull):
                        for i in range(len(convex_hull)-1,idx_t,-1):
                                path2.append(convex_hull[i])

        else:
                if idx_s != 0:
                        for i in range(idx_s-1,idx_t,-1):
                                path2.append(convex_hull[i])
        path2.append(t)

        for i in range(1,len(path1)):
                weight_1 += distance(path1[i-1],path1[i])
        for i in range(1,len(path2)):
                weight_2 += distance(path2[i-1],path2[i])

        return (WeightedPath(path=path1,weight=weight_1),
                WeightedPath(path=path2,weight=weight_2))
```

```
def division_conquista(s, t, safe_points):
        #ordenar los puntos por su componente x
        sorted_points = sorted(safe_points, key=get_key)

        result = divide(s,t,sorted_points)

        return result

if len(sys.argv) != 3 :
        print("Invalid input.\nUsage: \n\t safepath inputFilepath F|G|D")
        sys.exit(1)

filename = sys.argv[1]
mode=sys.argv[2]

#
#
convex_hull = []
path1 = None
path2 = None
s = ()
t = ()
safe_points = []

input = open(filename, "r")

for line in input:
        line = line.rstrip('\n')
        coordinates = line.split(" ")
        safe_points.append(Coordinate(x=int(coordinates[0]), y=int(coordinates[1])))

s = safe_points[0]
t = safe_points[1]

if mode == 'F':
        convex_hull = fuerza_bruta(s, t, safe_points)

        if s not in convex_hull or t not in convex_hull:
                print("ERROR: Convex Hull does not contain source or tail.")
                sys.exit(500)

        path1,path2=armar_caminos(s,t,convex_hull)

elif mode == 'G':
        convex_hull=grahamScan.convex_hull(safe_points)

        if s not in convex_hull or t not in convex_hull:
                print("ERROR: Convex Hull does not contain source or tail.")
                sys.exit(500)
```

```
        path1,path2=armar_caminos(s,t,convex_hull)
elif mode == 'D':
        convex_hull=division_conquista(s, t, safe_points)

        if s not in convex_hull or t not in convex_hull:
                print("ERROR: Convex Hull does not contain source or tail.")
                sys.exit(500)
        path1,path2=armar_caminos(s,t,convex_hull)
else:
        print("Method [%s] is not valid." % mode)
        os._exit(-1)

print("Camino 1: Longitud [%s]" % path1.weight)
print("Recorrido: " + str(path1.path))

print("Camino 2: Longitud [%s]" % path2.weight)
print("Recorrido: " + str(path2.path))

print("Camino seleccionado: [%s]" % (1 if path1.weight < path2.weight else 2))
```

- Apéndice - Graham Scan

```
from math import atan2

# para ordenar según su ángulo en radianes
def sort_by_angle(pivot, point):
    return atan2(point[1] - pivot[1], point[0] - pivot[0])

# determinante de P1P2 P1P3, define si el giro es a der o izq
def det(p1, p2, p3):
    return (p2[0] - p1[0]) * (p3[1] - p1[1]) - (p2[1] - p1[1]) * (p3[0] - p1[0])

def convex_hull(points):
    # selecciono el punto con la menor coordenada y, en caso de empate, el de menor coordenada x
    min_point = None
    for _, point in enumerate(points):
        if min_point is None or point.y < min_point.y:
            min_point = point
        if point.y == min_point.y and point.x < min_point.x:
            min_point = point

    # ordeno según su ángulo
    sorted_points = sorted(points, key=lambda x: sort_by_angle(min_point, x))
    sorted_points.remove(min_point)

    # el punto mínimo y el de menor ángulo tienen que pertenecer a la envoltura convexa
    hull = [min_point, sorted_points[0]]

    # por cada uno de los siguientes puntos ordenados, calculo si giro a derecha o izquierda
    # - si giro a izquierda, agrego el nuevo punto a la envoltura convexa
    # - mientras gire a derecha, remuevo el último punto de la envoltura convexa
    for point in sorted_points[1:]:
        while det(hull[-2], hull[-1], point) <= 0:
            if(len(hull) > 2):
                del hull[-1]
            else:
                break
        hull.append(point)
    return hull
```