Juan Carlos Díaz Rodríguez
Subgrupo U1
juancdia@ucm.es

Geometría Computacional
Práctica Opcional. Problema 4

04/04/2023

1. Introducción

En esta práctica estudiamos cómo detectar si dos segmentos del plano intersecan y una forma eficiente de hallar el número de componentes conexas en un grafo. Por último, veremos dos formas de abordar un problema NP-difícil, en concreto calcular el mínimo número de segmentos para obtener una única componente conexa.

2. Material utilizado

Dado que vamos a tratar un problema computacionalmente exigente, parte de la resolución de la práctica se ha hecho en C++, mientras que la parte de representación gráfica se ha mantenido en Python. Para pasar datos de un lenguaje a otro se guardan los datos en archivos de texto que leen o escriben ambos programas.

La resolución de ambos apartados tiene 4 partes diferenciadas.

En primer lugar, se requiere de un algoritmo para saber si dos segmentos intersecan. Se ha implementado la versión aquí explicada, además de una función para detectar cuando dos segmentos intersecan en los extremos. Si dos segmentos $\overline{AB}$ y $\overline{CD}$ intersecan en el "interior" de los segmentos, entonces la orientación de los puntos A, B, C será contraria a la de A, B, D y análogamente para el otro segmento. Dicha orientación se puede ver como el signo del coseno del ángulo que forman $\overline{AC}$ y $\overline{BC}$ para el primer caso. En caso de que la intersección se dé en un extremo, entonces, sin pérdida de generalidad, podemos decir que el punto C queda en el segmento $\overline{AB}$. Para comprobar esto basta tomar el producto vectorial como si $\overline{AB}$ y $\overline{AC}$ fuesen vectores de $\mathbb{R}^3$ en $z = 0$. Si el producto es 0 entonces los vectores son paralelos y basta ver que la componente x de C queda entre las componenetes x de los otros dos puntos.

Para hallar el número de componentes conexas se emplea una estructura de datos llamada `UnionFindForest` estudiada en una asignatura anterior. Calcular el número de componentes conexas se reduce a iniciar la estructura a N conjuntos disjuntos, recorrer los pares distintos de segmentos e ir comprobando si intersecan. Cuando intersequen dos segmentos se pide a la estructura que una sus componentes. Una serie de m operaciones de búsqueda, adición o unión en un `UnionFindForset` de n elementos tiene un coste amortizado de $\mathcal{O}(m\alpha(n))$ donde α es la función inversa de Ackermann.

Para el segundo apartado se resuelve el problema tanto con un algoritmo voraz como con la técnica de Ramificación y Poda (RyP). Par el primero, en cada iteración se une el segmento, de entre todos los posibles, que más componentes conexas una. Para el segundo enfoque es necesario poder obtener una cota inferior de los segmentos que nos van a hacer falta añadir a una solución parcial. Si suponemos que el segmento añadido en esa ocasión es el que más componenetes conexas unía, el siguiente que añadamos podrá, a lo sumo, unir una más que el anterior. Si denotamos por Y las componentes conexas que unió el mejor segmento y por X las componentes que nos quedan por unir, este problema se convierte en hallar el r tal que:

$$\sum_{k=1}^{r-1}(Y+k) < X \leq \sum_{k=1}^{r}(Y+k)$$

que es equivalente a resolver una ecuación de segundo grado y tomar la solución $r = \frac{-2Y-1+\sqrt{4Y^2+4Y+1+8X}}{2}$. Con lo que nuestra cota inferior será la suma de los segmentos que ya hayamos añadido más $\lceil r \rceil$. Para mejorar el rendimiento de RyP iniciamos la mejor solución a la del algoritmo voraz. Para que esta cota sea correcta hay que asegurarse de que no podamos meter en un futuro segmentos que unan más componentes que las consideradas por la cota optimista. Para ello llevaremos un conjunto de segmentos descartados que no consideraremos para extender la solución parcial. Esto elimina una gran cantidad de simetrías al obligar a las soluciones a considerar los segmentos en un orden concreto. Nótese que cualquier permutación de los segmentos sigue siendo el mismo conjunto de segmentos, pero el algoritmo lo consideraría soluciones distintas.

3. Resultados

Para dar una justificación del cambio de lenguaje, basta observar cómo Python, empleando exactamente el mismo algoritmo para calcular las componentes conexas, tarda 3.91 segundos en terminar, mientras que C++ tarda 17 milisegundos. Aunque la programación en C++ es ciertamente más compleja, dado que encontrar

una de las soluciones que ahora expondremos en C++ ha tardado 17 minutos, resulta completamente inviable estudiar nada de esta índole con Python.

Apartado i). Con el conjunto de segmentos propuesto se obtienen 338 componentes conexas, representadas en la Figura 1.

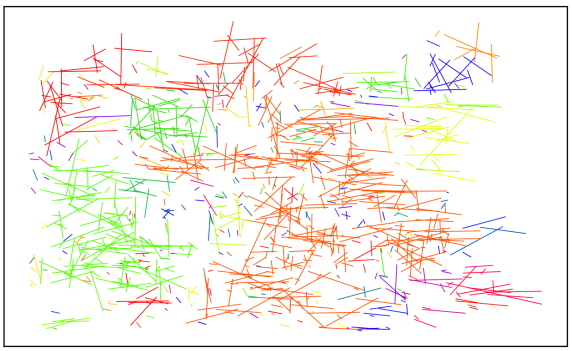

Figura 1: Componentes conexas iniciales

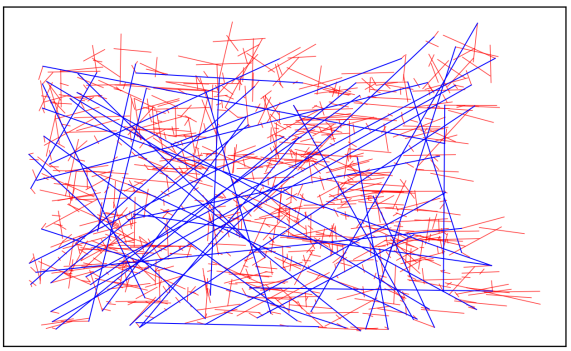

Figura 2: Solución voraz con 46 nuevos segmentos

Apartado ii). Como ya hemos comentado la resolución de este apartado es NP-difícil. Por ello se ha optado por una aproximación voraz. El número de segmentos hallado para conectar todas las componentes es de 46, como se puede ver en la Figura 2. En la Tabla 3 se muestran los tiempos que tarda el algoritmo voraz para distintos tamaños del conjunto de segmentos. La complejidad de este algoritmo es de $\mathcal{O}\left(an^3\alpha\left(n+a\right)\right)$, donde a es la cantidad de segmentos necesarios, según el algoritmo, para tener una única componente conexa.

Además de esta solución se ha programado el algoritmo de RyP. La anchura del árbol de soluciones es inabordable para tamaños superiores a 50. Para poder podar suficientes ramificaciones del árbol y evitar simetrías, cada solución parcial requiere una cantidad lineal de memoria, por lo que se alcanza rápidamente la cantidad de RAM que el ordenador puede manejar sin accesos constantes a memoria física. La buena noticia es que para todos los casos probados las soluciones dadas por el algoritmo voraz coinciden con la solución óptima calculada en RyP. Nótese que esto no quiere decir que las soluciones voraces sean óptimas en general. Esto se puede ver mejor en el ejemplo de la Figura 4, donde la secuencia de segmentos rojos podría ser una solución voraz, ya que al principio como máximo se pueden unir dos componentes con un único segmento, y la solución azul es una solución óptima.

N	Tiempo
10	1ms
20	7ms
40	63ms
80	553ms
150	4.01s
250	20.13s
350	62.17s
450	142.10s
550	267.73s
600	351.15s

Figura 3: Tiempos de ejecución

Aunque el algoritmo que ataca el problema NP-difícil no ha tenido mucho éxito, sí que nos ha permitido comprobar que la aproximación voraz, bajo unas condiciones de aleatoriedad se comporta extremadamente bien. Se incluye el código de ambos algoritmos en los archivos adjuntos.

4. Conclusión

De esta práctica podemos aprender lo siguiente:

- Hay formas elegantes de ver si dos segmentos intersecan basadas en geometría lineal básica.
- Es importante encontrar estructuras de datos eficientes para los problemas concretos a los que nos enfrentamos. Merece la pena buscarlas, pues hay multitud de estructuras muy eficientes con implementaciones sencillas.
- Hay que encontrar un equilibrio entre el tiempo de ejecución y el tiempo que lleve desarrollar un código. Aunque un lenguaje de programación sea muy sencillo de utilizar, puede suponer un detrimento de rendimiento no asumible para ciertos problemas
- En ocasiones es mejor renunciar a una solución óptima si se posee un algoritmo de aproximación con unas heurísticas bien dirigidas para el problema que acaece.

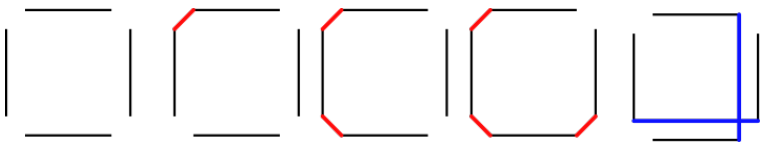

Figura 4: Contraejemplo de optimalidad

5. Código

Dado que el código en Python carece de mucho interés, pues o bien imita el de C++ o bien se usa únicamente para representar resultados, exponemos aquí el código de C++.

Se han omitido los detalles técnicos del lenguaje en relación a la declaración de estructuras, hashers y comparadores; así como la carga de datos y creación de segmentos aleatorios. Todo ello se puede encontrar en los archivos adjuntos.

Aquí se encuentra la estrucutra `UnionFind`:

```
//
//  UnionFind.h
//
//  Implementación de estructura para problemas de conectividad dinámica
//
//  Juan Carlos Díaz
//


#ifndef UNION_FIND_H
#define UNION_FIND

#include <vector>
#include <stack>
#include <stdexcept>


class UnionFind {
private:
  int _size; // Tamaño del grafo
  int _n_components;
  std::vector<int> _parent; // Array con los identificadores de las componentes
                            // conexas
  std::vector<int> _tree_size; // Mantiene el tamaño de cada árbol. Para mejora
                               // de coste

public:
  /**
   * Crea la estructura inicial con el grafo sin aristas
   */
  UnionFind(int size) :
    _size(size),
    _n_components(size),
    _parent(size),
    _tree_size(size)
  {
    // El padre de cada nodo es sí mismo
    for (int i = 0; i < size; ++i) _parent[i] = i;

    // El tamaño de cada árbol es 1
    std::fill(_tree_size.begin(), _tree_size.end(), 1);
  }

  /**
   * Une dos nodos dados
   * @throws domain_error si un nodo no está en el rango correcto
   */
  void node_union(int p, int q) {
    if (p < 0 || q < 0 || p >= _size || q >= _size)
      throw std::domain_error("node out of range");
    int i = find(p);
    int j = find(q);
```

```
  // Si ya pertenecen a la misma componente
  // conexa no hace falta hacer nada
  if (i == j) return;

  --_n_components;
  // Weighted quick-union
  // Para mantener el árbol lo más plano posible
  // siempre se añade el árbol de menos peso al de más
  if (_tree_size[i] < _tree_size[j]) {
    _parent[i] = _parent[j];
    _tree_size[j] += _tree_size[i];
  }
  else {
    _parent[j] = _parent[i];
    _tree_size[i] += _tree_size[j];
  }
}

void add_node(){
  ++_n_components;
  _tree_size.push_back(1);
  _parent.push_back(_size);
  ++_size;
}

/**
 * Devuelve el identificardor de la componente conexa a la que pertenece p
 */
int find(int p) {
  int root = p;
  std::stack<int> to_fix;
  while (_parent[root] != root) {
    to_fix.push(root);
    root = _parent[root];
  }
  // Path compression
  // Cada nodo que haya sido examinado se conecta directamente a la raíz
  while (!to_fix.empty()) {
    if (_parent[to_fix.top()] != root) {
      _tree_size[_parent[to_fix.top()]] -= _tree_size[to_fix.top()];
      _parent[to_fix.top()] = root;
    }
    to_fix.pop();
  }
  return root;
}

/**
 * Devuelve el tamaño de la componente conexa de p
 */
int size(int p) {
  int i = find(p);
  return _tree_size[i];
}

/**
 * Devuelve el número de componenets conexas
 */
int components() const {
  return _n_components;
}
```

```
  /**
   * Devuelve verdadero si p y q pertenecen a la misma componente conexa
   */
  bool connnected(int p, int q) {
    return find(p) == find(q);
  }

  /**
   * Devuelve la cantidad de nodos del grafo considerado
  */
  int nodes() const{
    return _size;
  }


};
#endif /* UNION_FIND_H */
```

A continuación se muestra el código relacionado con los algoritmos de intersección, voraz y RyP.

```
// ----------------------------
// Codigo para intersecciones
// ----------------------------

bool aligned(const Point &A, const Point &B, const Point &C){
    int ABx = B.x - A.x;
    int ABy = B.y - A.y;
    int ACx = C.x - A.x;
    int ACy = C.y - A.y;
    if (ABx*ACy-ACx*ABy == 0){
        return C.x <= max(A.x, B.x) && C.x >= min(A.x, B.x);
    }
    else{
        return false;
    }
}

bool coincide(const Segment &seg1, const Segment &seg2){
    return
        aligned(seg1.p1,seg1.p2,seg2.p1) ||
        aligned(seg1.p1,seg1.p2,seg2.p2) ||
        aligned(seg2.p1,seg2.p2,seg1.p1) ||
        aligned(seg2.p1,seg2.p2,seg1.p2);
}

#define ccw(A, B, C) ((C.y-A.y) * (B.x-A.x) > (B.y-A.y) * (C.x-A.x))

bool intersect(const Segment &seg1, const Segment &seg2){
    return  (ccw(seg1.p1,seg2.p1,seg2.p2) != ccw(seg1.p2,seg2.p1,seg2.p2)
            &&
            ccw(seg1.p1,seg1.p2,seg2.p1) != ccw(seg1.p1,seg1.p2,seg2.p2))
            ||
            coincide(seg1, seg2);
}

// -------
// Greddy
// -------

int joins_n_CC(
    const vector<Segment> &segs,
```

```
    const SegSet &added_segs,
    const Segment new_seg,
    UnionFind &CC
){
    int last_segment = CC.nodes();
    CC.add_node();
    int curr_CC = CC.components();
    const int N = segs.size();
    for (int i = 0; i < N; ++i){
        if (intersect(segs[i], new_seg)){
            CC.node_union(i, last_segment);
        }
    }
    for (auto seg_pair : added_segs){
        if (intersect(seg_pair.first, new_seg)){
            CC.node_union(seg_pair.second, last_segment);
        }
    }
    return curr_CC - CC.components();
}

void  update_greedy(
    const SegSet &new_segments,
    const vector<Segment> &segs,
    const Segment seg,
    const UnionFind &CC,
    int &best_value,
    Segment &best_seg)
{
    if (new_segments.count({seg, 0})) return;
    auto uf = UnionFind(CC);
    int solution = joins_n_CC(segs, new_segments, seg, uf);
    if (solution > best_value){
        best_value = solution;
        best_seg = seg;
    }
}
SegSet greedy(const vector<Segment> &segs, UnionFind CC){
    std::cout << "Starting with " << CC.components() << " componenets\n";
    const auto N = segs.size();
    SegSet new_segments(0);
    while(CC.components() > 1){
        int best_value = 0;
        Segment best_seg(0, 0, 0, 0);
        for (int i = 0; i < N; ++i){
            for (int j = 0; j < i; ++j){
                if (CC.find(i) != CC.find(j)){
                    // Try every pair of points
                    update_greedy(new_segments, segs, Segment(segs[i].p1, segs[j].p1), CC, best_value,
                    update_greedy(new_segments, segs, Segment(segs[i].p1, segs[j].p2), CC, best_value,
                    update_greedy(new_segments, segs, Segment(segs[i].p2, segs[j].p1), CC, best_value,
                    update_greedy(new_segments, segs, Segment(segs[i].p2, segs[j].p2), CC, best_value,
                }
            }
            if (i % 100 == 0){
                std::cout << "i = " << i << '\n';
            }
        }
        new_segments.insert({best_seg, CC.nodes()});
        joins_n_CC(segs, new_segments, best_seg, CC);
        std::cout << "Added: " << new_segments.size() << " segments. CC left: " << CC.components() <<
```

```
    }
    return new_segments;
}

// -----------------
// Branch and Bound
// -----------------

size_t cota_optimista(size_t added_already, size_t CC_left, size_t last_upgrade){
    if (added_already == 0){
        return INT_MAX - 1;
    }
    else{
        // Para saber cuántos segmentos hacen falta como mínimo hay que
        // tener en cuenta que si en un momento dado el mejor une Y componentes
        // conexas, a lo sumo el siguiente segmento puede unir Y + 1.
        // Si nos quedan X componenetes conexas esto supone hallar el r tal que:
/*
    \sum_{k=1}^{r-1} (Y+k) < x \leq \sum_{k=1}^{r} (Y+k)
    sii
    r^2-r+2Yr < 2x \leq r^2+r+2Yr
    Y para ello podemos hallar el ceil de la solución con radicando positivo de
    r = \frac{-2Y-1+\sqrt{4Y^2+4Y+1+8X}}{2}
*/
        float X = float(CC_left);
        float Y = float(last_upgrade);
        size_t solution = std::ceil(0.5*(-2*Y-1+std::sqrt(4*Y*(Y+1)+1+8*X)));
        return added_already + solution;
    }
}

struct solucion {

    UnionFind CC;
    SegSet added_segs;
    SegSet forbidden_segs;
    int coste_estimado; // cota superior de la mejor solución alcanzable
                        // desde esta solución parcial
};

struct Candidate{
    UnionFind CC;
    Segment addition;
    int bound;
};

SegSet branch_and_bound(
    const vector<Point>& points,
    const vector<Segment> &segs,
    UnionFind firstCC,
    int bound=-1){
    // Iniciar las soluciones con una única solución del voraz
    SegSet set_solution = greedy(segs, UnionFind(firstCC));
    size_t best_solution = set_solution.size();
    priority_queue<solucion, vector<solucion>, ComparaSoluciones> q;
    solucion start(UnionFind(firstCC), SegSet(0), SegSet(0), 0);
    q.push(start);
    while (!q.empty() && q.top().coste_estimado < best_solution){
        // Considerar los N^2 segmentos posibles
        // Para cada uno se mira si es prometedor, es decir,
        // si optimista < mejor_solucion. Para mirar si es prometedor ya se ha
```

```
// calculado el UF. Después se mete en el array que se ordenará después
// Una vez se tiene el array ordenado se van considerando si se coge o
// no y se mete en la cola añadiendo los nodos no cogidos como segmentos
// prohibidos. Esto debería quitar muchísimas simetrías.
solucion actual = q.top();
std::cout << "q: " << q.size() << "\n";
std::cout << "cota: " << actual.coste_estimado << "\n";
std::cout << "mejor: " << best_solution << "\n\n";
q.pop();
// Se comprueba si ya es solución
if (actual.CC.components() == 1){
    set_solution = SegSet(actual.added_segs);
    best_solution = set_solution.size();
    continue;
}
vector<size_t> indices;
vector<Candidate> candidates;
// Recorrer los segmentos
for (auto i = 0; i < points.size() - 1; ++i){
    for (auto j = i+1; j < points.size(); ++j){
        Segment newSeg = {points[i], points[j]};
        if (actual.forbidden_segs.count({newSeg, 0}))
            continue;
        if (actual.CC.find(i / 2) != actual.CC.find(j / 2)){
            // Mirar si es prometedor
            UnionFind newCC(actual.CC);
            // Con esta función vemos cuántas componenetes
            // conexas se unen y además obtenemos un UnionFind actualizado
            int last_upgrade = joins_n_CC(
                                    segs,
                                    actual.added_segs,
                                    {points[i], points[j]},
                                    newCC);

            int bound = cota_optimista(
                    actual.added_segs.size() + 1,
                    newCC.components(),
                    last_upgrade);
            if (bound < best_solution)
            {
                // Si es prometedor lo añadimos a los candidatos
                indices.push_back(indices.size());
                candidates.push_back({
                    UnionFind(newCC),
                    {points[i], points[j]},
                    bound});
            }
        }
    }
}
// Es necesario ordenar para preservar la veracidad de las cotas
sort(candidates.begin(), candidates.end(),
[](const Candidate& a, const Candidate& b){
    return a.bound < b.bound;
});
for (auto candidate : candidates){
    q.push({
        UnionFind(candidate.CC),
        SegSet(actual.added_segs),
        candidate.addition,
        SegSet(actual.forbidden_segs),
```

```
                candidate.bound
            });
            // Las siguiente soluciones tendrán prohibido emplear
            // los anteriores nodos. De esta forma se preserva
            // la veracidad de la cota
            actual.forbidden_segs.insert({candidate.addition, 0});
        }
    }
    return set_solution;
}
```