Apertura: domingo, 3 de noviembre de 2024, 08:03
Cierre: jueves, 12 de diciembre de 2024, 16:30

Objetivo

El objetivo principal de esta práctica consiste en **desarrollar un ejemplo de algoritmo que utilice la técnica algorítmica general *Voraz* (*Greedy*)** para obtener un **Euclidean minimum spanning tree (EMST) de un conjunto de puntos** en dos dimensiones (2D). Para ello se detalla a continuación una introducción al problema, los requisitos funcionales mínimos y opcionales, ejemplos de uso y los criterios de evaluación.

Introducción

Un árbol generador mínimo euclidiano de un conjunto finito de puntos en el plano euclidiano (2D) conecta los puntos mediante segmentos, minimizando la longitud total de los segmentos. En él, cualquier par de puntos puede alcanzarse mutuamente a lo largo de un camino que siga estos segmentos.

Se trata de un problema muy similar al conocido problema de optimización en grafos Árbol Generador Mínimo. De hecho, para su resolución vamos a implementar una versión adaptada del algoritmo de Kruskal para este problema.

El objetivo de la práctica es implementar la resolución del problema del Árbol Generador Mínimo en el Plano Euclídeo a través de un enfoque *Voraz*. Con ese propósito, se debe:

- Elaborar los siguiente tipos de datos: un tipo de datos que represente un punto (point), como par de *double*; un tipo *arco* como un par de elementos de tipo punto; un tipo de dato *arco ponderado*, como un par cuyo primer elemento es *double* y el segundo es tipo *punto*; un *vector de arcos ponderados*, como un vector cuyos elementos son *arcos ponderados*; un tipo *colección de puntos*, como un conjunto de *puntos*; y finalmente un tipo *árbol*, como un vector de *arcos*.

A continuación se muestra un ejemplo de definición de los tipos de dato:

```
#pragma once

#include <iostream>
#include <utility>
#include <vector>
#include <set>

#define MAX_SZ 3
#define MAX_PREC 0

namespace CyA
{
    typedef std::pair<double, double> point;

    typedef std::pair<point, point> line;
    typedef std::vector<point> point_vector;

    typedef std::pair<point, point> arc;
    typedef std::pair<double, arc> weigthed_arc;
    typedef std::vector<weigthed_arc> arc_vector;

    typedef std::set<point> point_collection;

    typedef std::vector<arc> tree;
}

std::ostream& operator<<(std::ostream& os, const CyA::point_vector& ps);
std::ostream& operator<<(std::ostream& os, const CyA::point& ps);

std::istream& operator>>(std::istream& is, CyA::point_vector& ps);
std::istream& operator>>(std::istream& is, CyA::point& ps);
```

Así como de su implementación:

```cpp
#include "point_types.hpp"

#include <iomanip>

std::ostream& operator<<(std::ostream& os, const CyA::point_vector& ps)
{
    os << ps.size() << std::endl;

    for (const CyA::point &p : ps)
    {
        os << p << std::endl;
    }

    return os;
}

std::ostream& operator<<(std::ostream& os, const CyA::point& p)
{
    os << std::setw(MAX_SZ) << std::fixed << std::setprecision(MAX_PREC) << p.first << "\t" << std::setw(MAX_SZ) <<
std::fixed << std::setprecision(MAX_PREC) << p.second;

    return os;
}

std::istream& operator>>(std::istream& is, CyA::point_vector& ps)
{
    int n;
    is >> n;

    ps.clear();

    for (int i = 0; i < n; ++i)
    {
        CyA::point p;
        is >> p;

        ps.push_back(p);
    }

    return is;
}

std::istream& operator>>(std::istream& is, CyA::point& p)
{
    is >> p.first >> p.second;

    return is;
}
```

- A continuación debe definirse e implementarse una clase *sub árbol* que contendrá un árbol, una colección de puntos y el coste de dicho sub árbol. Se muestra un ejemplo de definición:

```cpp
#pragma once

#include <iostream>
#include <cmath>

#include "point_types.hpp"

namespace EMST
{
    class sub_tree
    {
    private:
        CyA::tree arcs_;
        CyA::point_collection points_;
        double cost_;

    public:
        sub_tree(void);
        ~sub_tree(void);

        void add_arc(const CyA::arc &a);
        void add_point(const CyA::point &p);
        bool contains(const CyA::point &p) const;
        void merge(const sub_tree &st, const CyA::weigthed_arc &a);

        inline const CyA::tree& get_arcs(void) const { return arcs_; }
        inline double get_cost(void) const { return cost_; }
    };

    typedef std::vector<sub_tree> sub_tree_vector;
}
```

Así como su implementación:

```cpp
#include <iomanip>
#include <cmath>

#include <algorithm>
#include <utility>

namespace EMST
{
    sub_tree::sub_tree(void) : arcs_(),
                               points_(),
                               cost_(0)

    {
    }

    sub_tree::~sub_tree(void)
    {
    }

    void sub_tree::add_arc(const CyA::arc &a)
    {
        arcs_.push_back(a);

        points_.insert(a.first);
        points_.insert(a.second);
    }

    void sub_tree::add_point(const CyA::point &p)
    {
        points_.insert(p);
    }

    bool sub_tree::contains(const CyA::point &p) const
    {
        return points_.find(p) != points_.end();
    }

    void sub_tree::merge(const sub_tree &st, const CyA::weigthed_arc &a)
    {
        arcs_.insert(arcs_.end(), st.arcs_.begin(), st.arcs_.end());
        arcs_.push_back(a.second);

        points_.insert(st.points_.begin(), st.points_.end());

        cost_ += a.first + st.get_cost();
    }
}
```

Los métodos de esta clase deben permitir añadir un arco o un vértice al sub árbol, permitir determinar si un punto se encuentra en un sub árbol, o incluso unir un sub árbol (invocante) con otro pasado por parámetro, a través de un arco.

A partir de estos elementos se debe implementar el algoritmo de Kruskal para una entrada determinada:

Definición del Algoritmo de Kruskal:

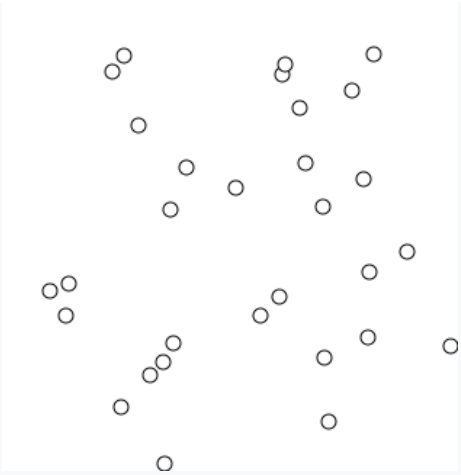
1. Crear un bosque (F) donde cada vértice en el grafo es un árbol separado.
2. Crear un conjunto ordenado (S) que contenga todas las aristas del grafo.
3. Mientras (S) no esté vacío y (F) aún no abarque todo el grafo:

- Eliminar una arista con el peso mínimo de (S).
- Si la arista eliminada conecta dos árboles diferentes, entonces añadirla al bosque (F), combinando dos árboles en uno solo.

Al finalizar el algoritmo, el bosque forma un bosque generador mínimo del grafo. Si el grafo está conectado, el bosque tendrá un solo componente y formará un árbol generador mínimo.

Completando la Definición:

4. Durante el proceso, el algoritmo asegura que no se formen ciclos al añadir una nueva arista, ya que solo une árboles separados.
5. El algoritmo se detiene cuando todas las aristas han sido examinadas o cuando el bosque (F) ha alcanzado el estado de abarcar todos los vértices del grafo, lo que sucede primero.
6. La eficiencia del algoritmo depende del método utilizado para seleccionar la arista de peso mínimo y para verificar la conexión entre árboles en el bosque.



El siguiente fragmento de código contiene la lógica del algoritmo anteriormente definido:

```
    void point_set::EMST(void)
    {
        CyA::arc_vector av;
        compute_arc_vector(av);

        forest st;

        for (const CyA::point &p : *this)
        {
            sub_tree s;
            s.add_point(p);

            st.push_back(s);
        }

        for (const CyA::weigthed_arc &a : av)
        {
            int i, j;
            find_incident_subtrees(st, a.second, i, j);

            if (i != j)
            {
                merge_subtrees(st, a.second, i, j);
            }
        }

        emst_ = st[0].get_arcs();
    }
```

Este método corresponde a la siguiente clase:

```
typedef std::vector<sub_tree> forest;

    class point_set : public CyA::point_vector
    {
    private:
        CyA::tree emst_;

    public:
        point_set(const CyA::point_vector &points);
        ~point_set(void);

        void EMST(void);

        void write_tree(std::ostream &os) const;
        void write(std::ostream &os) const;

        inline const CyA::tree& get_tree(void) const { return emst_; }
        inline const CyA::point_vector& get_points(void) const { return *this; }
        inline const double get_cost(void) const { return compute_cost(); }

    private:
        void compute_arc_vector(CyA::arc_vector &av) const;
        void find_incident_subtrees(const forest& st, const CyA::arc &a, int& i, int& j) const;
        void merge_subtrees(forest& st, const CyA::arc &a, int i, int j) const;

        double compute_cost(void) const;

        double euclidean_distance(const CyA::arc& a) const;
    };
```

A continuación se muestra el código para calcular los costes entre cada par de elementos:

```
void point_set::compute_arc_vector(CyA::arc_vector &av) const
    {
        av.clear();

        const int n = size();

        for (int i = 0; i < n - 1; ++i)
        {
            const CyA::point &p_i = (*this)[i];

            for (int j = i + 1; j < n; ++j)
            {
                const CyA::point &p_j = (*this)[j];

                const double dist = euclidean_distance(std::make_pair(p_i, p_j));

                av.push_back(std::make_pair(dist, std::make_pair(p_i, p_j)));
            }
        }

        std::sort(av.begin(), av.end());
    }
```

Requisitos funcionales mínimos

Para superar la práctica, los requisitos funcionales mínimos exigidos son:

- Implementar el enfoque *voraz* para el cálculo del EMSP a través del algoritmo de Kruskal adaptado.
- Obtener el resultado de dicha resolución a través del conjunto de aristas que conforman el árbol.
- La clases debe ser ajustada lo necesario para adaptarse al estilo exigido en las prácticas, Google C++ Style Guide
- El programa realizado ha de ejecutarse leyendo desde la entrada estándar un conjunto de puntos, y emitiendo su resultado (conjunto de aristas) a la salida estándar

Requisitos funcionales opcionales

- Añadir la opción -d que genere un archivo DOT para la posterior visualización del árbol

Ejemplo Graphviz

```
graph {
    a [pos="0,0!"]
    b [pos="0,1!"]
    c [pos="2,2!"]
    d [pos="-1,0!"]
    a -- b
    a -- c
    a -- d
  }
```

Línea de comando:

```
neato ejemplo.dot -Tpdf -o salida.pdf
```

Ejemplos de prueba

Algunos ejemplos de entradas y sus posibles salidas están disponibles en esta carpeta

Evaluación

Se evaluará positivamente los siguientes aspectos:

- Funcionamiento correcto de la **modificación solicitada**: si no se consigue realizar la modificación la práctica será valorada con un cero.
- Presentación en el laboratorio: el **grado de funcionamiento de la práctica**, y si desarrolla **requisitos opcionales** o **mejoras significativas**.
- Código subido en la tarea correspondiente a la práctica: **buen diseño**, **estructura de clases** y **limpieza**.

Teniendo superados los aspectos anteriormente descritos, una evaluación orientativa sería la siguiente:

- Si se desarrolla el algoritmo y se muestra la solución simple: **hasta 5 puntos**.
- Si se desarrolla una solución más elaborada: **hasta 7 puntos**.

- Si se desarrolla el requisito funcional opcional, y además se ha desarrollado un algoritmo más eficiente y directa, valorándose la sencillez (en longitud y comprensión) del diseño propuesto: **hasta 10 puntos**.

Profesor responsable

Si surgiera alguna **duda más sobre la práctica, el enunciado o la corrección**, podría formularse a través del correo electrónico del profesor **Jorge Riera** (grupo de mañana) o del correo electrónico del profesor **José Luis González Ávila** (grupo de tarde).

Agregar entrega

Estado de la entrega

Estado de la entrega	Todavía no se han realizado envíos
Estado de la calificación	Sin calificar
Tiempo restante	6 días 7 horas restante
Última modificación	-
Comentarios de la entrega	Comentarios (0)