Brandon Toner

September 16, 2021

CSCI 405: Algorithm Analysis II

Homework 6: Polygon Decomposition

Code

```
#include <errno.h>
#include <math.h>
#include <stdarg.h>
#include <stdio.h>
#include <string.h>
#include <stdexcept>
#include <vector>

using std::vector;
using std::logic_error;
void LogicError(const char* format, ...) {
    char buffer[4096];
    va_list args;
    va_start(args, format);
    vsprintf(buffer, format, args);
    va_end(args);
    throw logic_error(buffer);
}

class Vertex {
  private:
    double x_;
    double y_;
    size_t index_;

  public:
    Vertex(const double x, const double y, const size_t index)
    : x_(x), y_(y), index_(index) { }
    double x() const {
```

```
        return x_;
    }
    double y() const {
        return y_;
    }
    size_t index() const {
        return index_;
    }
    static double distance(const Vertex& a, const Vertex& b) {
        double dx = a.x_ - b.x_;
        double dy = a.y_ - b.y_;
        return sqrt(dx * dx + dy * dy);
    }
} ;

class Chord {
  private:
    const Vertex* one_;
    const Vertex* two_;
    double dist_;

  public:
    Chord() : one_(NULL), two_(NULL), dist_(INFINITY) { }
    Chord(const Vertex& one, const Vertex& two)
    : one_((one.index() < two.index()) ? (&one) : (&two)),
      two_((one.index() < two.index()) ? (&two) : (&one)),
      dist_(Vertex::distance(one, two)) { }
    double dist() const {
        return dist_;
    }
    const Vertex* one() const {
        return one_;
    }
    const Vertex* two() const {
        return two_;
```

```
    }
    void print() const {
        printf("%3lu:(%12.4f,%12.4f) => %3lu:(%12.4f,%12.4f) dist: %12.4f\n",
               one_->index(), one_->x(), one_->y(),
               two_->index(), two_->x(), two_->y(),
               dist_);
    }
    bool operator ==(const Chord& other) const {
        return (one_->index() == other.one()->index() &&
                two_->index() == other.two()->index());
    }
} ;

class Polygon {
  private:
    const char*     path_;
    vector<Vertex>  points_;
    vector<Chord>   chords_;
    double**        value_matrix_;
    double**        dists_matrix_;
    vector<Chord>** chord_matrix_;
    void read() {
        FILE* file;
        if ((file = fopen(path_, "r")) == NULL)
            LogicError("fopen('%s', 'r') error: %s", path_, strerror(errno));
        double x, y;
        while (fscanf(file, "%lf%lf", &x, &y) == 2)
            points_.push_back(Vertex(x, y, points_.size()));
        fclose(file);
    }
    void add_chord(const Chord& c, vector<Chord> * const out) {
        // Don't add if edge.
        if ((c.one()->index() + 1) % points_.size() == c.two()->index() ||
            (c.two()->index() + 1) % points_.size() == c.one()->index())
            return;
```

```
    // Don't add if chord exists in chords_
    for (auto i = out->begin(); i != out->end(); ++i)
        if (c == *i) return;
    out->push_back(c);
}
static void copy_to(vector<Chord> * const src, vector<Chord> * const dst) {
    for (auto i = src->begin(); i != src->end(); ++i)
        dst->push_back(*i);
}
double triangle_perim(size_t i, size_t j, size_t k) {
    return dists_matrix_[i][j] + dists_matrix_[j][k] +
           dists_matrix_[k][i];
}
double decompose(const size_t i, const size_t j, vector<Chord>* const out) {
    if (value_matrix_[i][j] >= 0) {
        copy_to(&chord_matrix_[i][j], out);
        return value_matrix_[i][j];
    } else if (j == i + 1) {
        return value_matrix_[i][j] = 0;
    } else {
        double min = INFINITY;
        size_t best_k;
        vector<Chord> best_out;
        for (size_t k = i + 1; k < j; ++k) {
            vector<Chord> new_out;
            double v = decompose(i, k, &new_out)
                     + decompose(k, j, &new_out)
                     + triangle_perim(i, j, k);
            if (v < min) {
                min = v;
                best_k = k;
                best_out = new_out;
            }
        }
        add_chord(Chord(points_[i],      points_[j]),      &best_out);
```

```
            add_chord(Chord(points_[j],      points_[best_k]), &best_out);
            add_chord(Chord(points_[best_k], points_[i]),      &best_out);
            copy_to(&best_out, out);
            chord_matrix_[i][j]   = best_out;
            return value_matrix_[i][j] = min;
        }
    }

  public:
    explicit Polygon(const char* path) : path_(path) {
        read();
        value_matrix_ = new double*[points_.size()];
        dists_matrix_ = new double*[points_.size()];
        chord_matrix_ = new vector<Chord>*[points_.size()];
        for (size_t i = 0; i < points_.size(); ++i) {
            value_matrix_[i] = new double[points_.size()];
            dists_matrix_[i] = new double[points_.size()];
            chord_matrix_[i] = new vector<Chord>[points_.size()];
            for (size_t j = 0; j < points_.size(); ++j) {
                value_matrix_[i][j] = -1;
                dists_matrix_[i][j] = Vertex::distance(points_[i], points_[j]);
            }
        }
    }
    double decompose() {
        return decompose(0, points_.size() - 1, &chords_);
    }
    void print_chords() {
        for (auto i = chords_.begin(); i != chords_.end(); ++i)
            i->print();
    }
} ;
int main() {
    Polygon p("polygon2.txt");
    printf("size: %f\n", p.decompose());
```

```
    p.print_chords();
}
```

Results

```
size: 8328.604014
  0:(    397.2494,    204.0564) =>   2:(    375.8121,    295.3134) dist:      93.7411
  3:(    340.3170,    338.5171) =>   5:(    260.6411,    384.6494) dist:      92.0676
  2:(    375.8121,    295.3134) =>   5:(    260.6411,    384.6494) dist:     145.7576
  5:(    260.6411,    384.6494) =>   7:(    163.0483,    394.1319) dist:      98.0524
  7:(    163.0483,    394.1319) =>   9:(     76.2607,    352.6056) dist:      96.2108
  5:(    260.6411,    384.6494) =>   9:(     76.2607,    352.6056) dist:     187.1442
  9:(     76.2607,    352.6056) =>  11:(      8.0842,    273.5640) dist:     104.3820
 11:(      8.0842,    273.5640) =>  13:(      8.6748,    173.7644) dist:      99.8013
  9:(     76.2607,    352.6056) =>  13:(      8.6748,    173.7644) dist:     191.1858
  5:(    260.6411,    384.6494) =>  13:(      8.6748,    173.7644) dist:     328.5719
 13:(      8.6748,    173.7644) =>  15:(     34.3564,     87.0327) dist:      90.4540
 13:(      8.6748,    173.7644) =>  16:(     72.7005,     46.8978) dist:     142.1071
 17:(    117.8008,     12.5129) =>  19:(    210.7204,      2.7835) dist:      93.4276
 16:(     72.7005,     46.8978) =>  19:(    210.7204,      2.7835) dist:     144.8985
 13:(      8.6748,    173.7644) =>  19:(    210.7204,      2.7835) dist:     264.6826
 19:(    210.7204,      2.7835) =>  21:(    309.2761,     27.5857) dist:     101.6286
 21:(    309.2761,     27.5857) =>  23:(    377.3673,    108.9847) dist:     106.1236
 19:(    210.7204,      2.7835) =>  23:(    377.3673,    108.9847) dist:     197.6104
 13:(      8.6748,    173.7644) =>  23:(    377.3673,    108.9847) dist:     374.3402
  5:(    260.6411,    384.6494) =>  23:(    377.3673,    108.9847) dist:     299.3594
  2:(    375.8121,    295.3134) =>  23:(    377.3673,    108.9847) dist:     186.3352
  0:(    397.2494,    204.0564) =>  23:(    377.3673,    108.9847) dist:      97.1284
```

Justification of approx. $\Theta(n^3)$ running time

The algorithm builds a tableau of the best polygon decompositions for as it runs, which takes linear time to fill each of the entries. Since there are n^2 entries, it should take $\Theta(n^3)$ running time.