Assessing The Effectiveness of an Original Heuristic Algorithm for Computing the Asymmetric Travelling Salesman Problem

Research question: To what extent is the polygon expansion algorithm a suitable heuristic for computing an approximate solution for the asymmetric travelling salesman problem?

Subject: Computer Science

Word Count: 2006

Table of Contents

[1. Introduction [done] 3](#_Toc94566120)

[2. Background information on the problem [done] 3](#_Toc94566121)

[a. Travelling salesman problem 3](#_Toc94566122)

[b. Asymmetric travelling salesman problem 5](#_Toc94566123)

[c. Exact algorithm 5](#_Toc94566124)

[3. The polygon expansion algorithm 6](#_Toc94566125)

[a. Mathematical Vectors 6](#_Toc94566126)

[b. The Polygon 7](#_Toc94566127)

[c. Vector replacement 8](#_Toc94566128)

[d. Selection criteria for expanding edges 9](#_Toc94566129)

[e. Special distances 11](#_Toc94566130)

[4. Implementation and analysis of the algorithm 11](#_Toc94566131)

[a. Handling infinity 11](#_Toc94566132)

[b. Storing values in a dynamically allocated matrix 12](#_Toc94566133)

[c. Polygon expansion 12](#_Toc94566134)

[5. Methodology 14](#_Toc94566135)

[6. Data processing 15](#_Toc94566136)

[7. Results discussion and conclusion 15](#_Toc94566137)

[Bibliography 16](#_Toc94566138)

# Introduction

The travelling salesman problem is arguably the most studied optimization problem in the world.[[1]](#footnote-1) It is an NP-hard problem that has puzzled mathematicians and computer scientists alike for over 90 years since it was initially formulated in 1930. There has yet to be a polynomial-time algorithm for the exact algorithm of the general-case problem.[[2]](#footnote-2) The travelling salesman problem has many applications in real life, from optimizing logistics to DNA sequencing.[[3]](#footnote-3)

This paper will be exploring a particular variant of the travelling salesman problem in detail – the asymmetric travelling salesman problem. It will give an original algorithm, explain the algorithm, and evaluate its effectiveness through the following metric: the accuracy of the algorithm, answering the topic: Evaluating the Polygon Expansion Algorithm as a Viable Heuristic for the Asymmetric Travelling Salesman Problem?

All terms will be explored more thoroughly in the following sections of the exploration.

# Background information on the problem

## Travelling salesman problem

The travelling salesman problem in essence is a problem featuring a salesman trying to find the shortest route that it would need to take to tour all designated cities on a map.[[4]](#footnote-4) The distances between these cities are predetermined, our salesman friend must visit all of the cities on the map, and he cannot revisit cities.

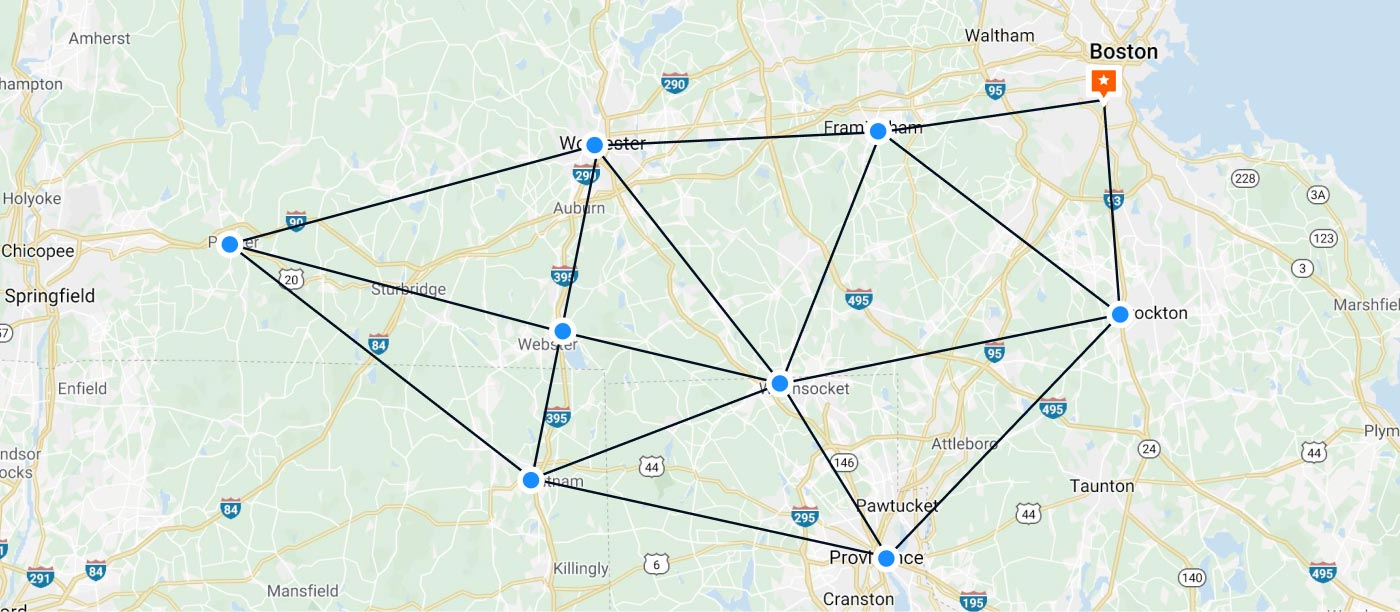


Figure 1: Example of the travelling salesman problem[[5]](#footnote-5)

Let us number all the cities that the salesman must cover 0 to n, in which n + 1 represents the number of cities that the salesman has to cover in his tour. According to the definition of the problem, the salesman will have to scale all n + 1 cities for a full tour.

The “map” that the salesman will be travelling on will be represented as an (n + 1) × (n + 1) matrix. Let D[j][k] denote an entry on the jth row and kth column of this matrix. Each entry D[j][k] will represent the distance between the jth city and the kth city. Due to the nature of the original travelling salesman problem, D[j][k] = D[k][j].

The other crucial half of the problem is the route that the salesman will take in his travels. This can be modelled through the use of a list of numbers. The list will detail the sequence in which the salesman has travelled in sequence. Incomplete lists would thus represent states in which the salesman has not yet toured all cities.

In the travelling salesman problem, a road between two cities may not exist. In this case, it will be replaced with an infinite value.

## Asymmetric travelling salesman problem

The asymmetric travelling salesman problem is a variant of the problem in which D[j][k] need not equal D[k][j]. A real-life analogue of this scenario would be if D[j][k] denoted the time our trusty salesman needed to scale the distance between city j and city k. If he needs to travel uphill from city j to city k, his time will be longer than when he comes back, going downhill from city k to city j.

Modelling the problem will be analogous to modelling the normal travelling salesman problem, except that the (n + 1) × (n + 1) matrix will not have any restrictions as to which entries can be added into the matrix.

## Exact algorithm

The asymmetric travelling salesman problem can be solved by designating a starting city, recursively exploring the rest of the cities and comparing them. However, this algorithm has a best and worst-case complexity of O(n!). This number rises very rapidly and exploring all cases would take an unimaginably long amount of time, even for relatively small cases less than 50.[[6]](#footnote-6) Thus, to solve the problem for larger cases, heuristic, or approximate algorithms are used.

# The polygon expansion algorithm

The polygon expansion algorithm is an original heuristic algorithm that I have devised to calculate an approximate solution to the travelling salesman problem. However, to better understand the algorithm mandates the understanding of certain mathematical objects.

## Mathematical Vectors

A mathematical vector is a mathematical object with both a magnitude (or “length”) and a direction. This mathematical object can be expressed using an arrow pointing in a specific direction.[[7]](#footnote-7) Vectors can be added together to form a new vector. This operation follows the rule such that the vector sum of two initial vectors is obtained through placing the two initial vectors head to tail and drawing the vector from the free tail to the free head as shown below. [[8]](#footnote-8) Diagram

Description automatically generated

Figure 3: the addition process of two vectors A and B.

The resultant vector is considered to be equivalent to the sum of the two initial vectors and can be used to substitute one another. This property is crucial in the algorithm and will be further expanded upon in section c.

## The Polygon

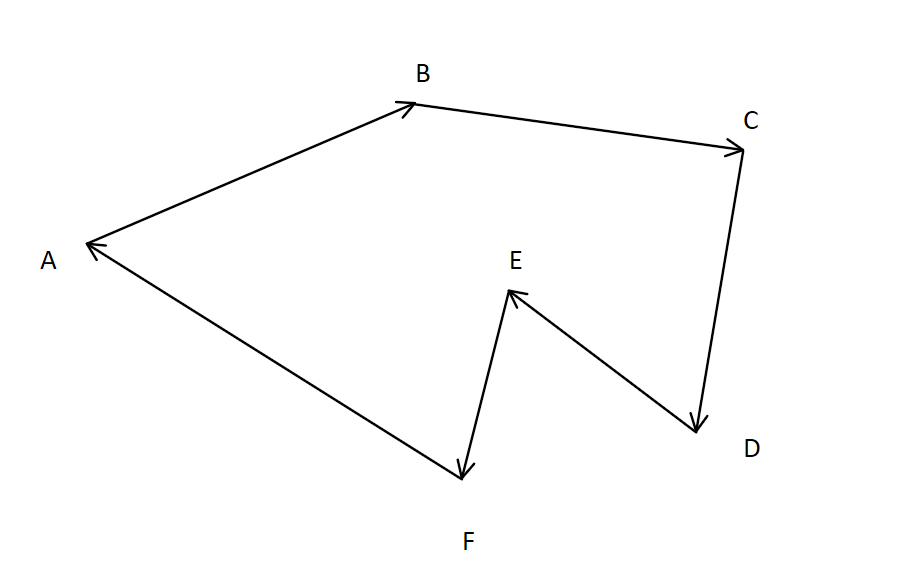
In this exploration, a closed polygon will be defined as a closed shape with rigid edges that are vectors placed head to tail, summing to 0 as shown below. This is due to the intrinsic nature of the asymmetric travelling salesman problem; the path that the salesman travelled cannot be reversed, and thus must be expressed as a series of vectors.

Figure 2: Example of a polygon

For the above instance, our salesman has taken the route A→B→C→D→E→F→A. Notice how the vectors constituting the polygon are connected head to tail, thus summing up to 0.

## Vector replacement

The central idea of the algorithm is that for any vector , we can substitute with vector . As shown in the diagram below. Chart, line chart

Description automatically generated

Figure 3: An example of replacing a vector with two others

As mentioned in the previous section, a tour of cities can be expressed as a polygon made up of vectors summing up to zero. As the two new vectors sum up to be the old vector, the new vector paths still sum up to zero, as they still form a closed path and all vectors are still connected head to tail. Through a single replacement, we can expand a polygon of vertex count k to a polygon of vertex count of k + 1.

Utilizing this property, for each replacement made to the polygon, an additional city will be added to the tour. When the number of cities reaches n + 1 (n + 1 being the total number of cities in the problem) the algorithm ends, and the total distance of the tour can be calculated.

For the sake of simplicity, one such replacement will be referred to as the “expansion of an edge” and the difference in magnitude of the two initial vectors and the resultant vector, in the example, will be called an “incremental value”

## Selection criteria for expanding edges

During the replacement process, there are multiple different choices for an edge of a polygon to expand to since there may be multiple unexplored cities for an incomplete tour.

The algorithm will need to evaluate all of these potential points and record the minimum incremental distance for all of them. As shown in the example below, the algorithm will record the red path as the new increment for the edge that is being explored.

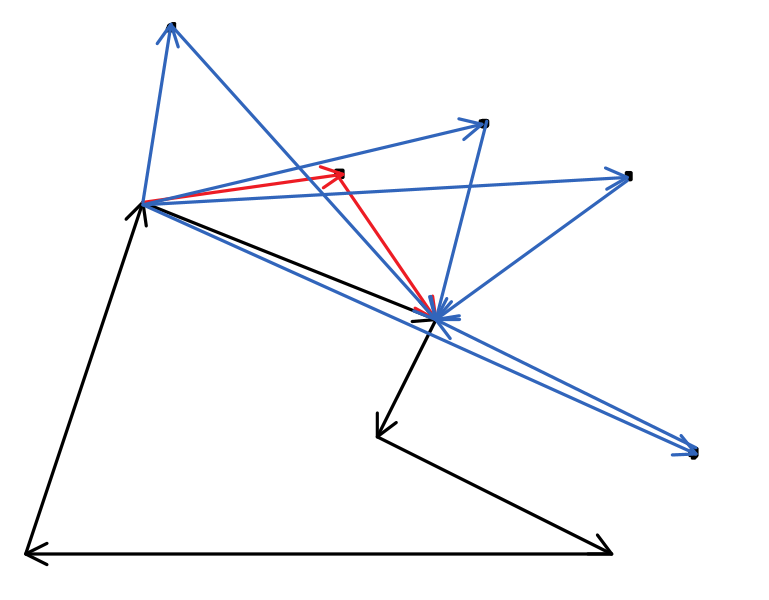


Figure 4: Selecting the most optimal route for one edge

This process will be repeated for all edges and the vertex with the minimum incremental distance will be recorded.

After all edges are explored, changes will be committed and the new polygon will be analyzed for the best change to be committed.

## Special distances

A road does not need to be available between two cities in the travelling salesman problem. To address this case, the algorithm will treat the distance of a non-existent road between two cities as infinite.

# Implementation and analysis of the algorithm

## Handling infinity

As there is no pre-existing data structure to handle infinity in the C++ standard library, I needed to implement my own data type, named in the program as an infinite double, or infDouble for short.

Text

Description automatically generated

Figure 5: Implementation of the Infinite double data type

The data type contains a double and an integer. The double value stores the non-infinite value of the number and can be added and subtracted from in the exact fashion one would from a regular double. The integer value would represent the number of infinite values that the number holds. The reason that the number of infinities can vary is to help with the calculation of incremental values, as an incremental value established on a non-existent edge may have a negative infinite value.

## Storing values in a dynamically allocated matrix

To facilitate the storage of values, I have constructed an object called a matrix, which is, in essence, a dynamically allocated 2-d array of objects. This object is used in two instances, the first being the storage of the input values, the second being the storage of all of the incremental values of the edges.

Worthy of notice is that the direction of the vectors is encoded in the matrix representation itself. For an entry D[j][k], the direction of the vector would be represented as from point j to point k with the magnitude being encoded in the matrix entry.

## Polygon expansion

The implementation of the polygon expansion algorithm is largely loyal to the algorithm itself. During the coding of the algorithm, I have noticed that a polygon in this instance does not necessarily need to have more than 3 edges. Two opposite vectors pointing at each other could be also considered to be a polygon.



Figure 6: The procedure for adding the best city to the tour

By repeatedly calling this procedure, we can expand the polygon until it covers all of the cities in the tour.

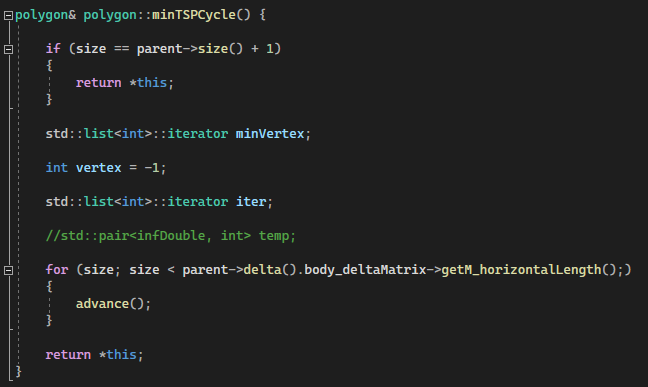


Figure 7: The code for expanding the polygon until the tour is complete

After calculating all values for all attempted initial polygons, we can find the minimal approximate value for the tour.

# Methodology

The experiment, written and executed in C++ will be addressed below.

The procedure for the experiment for the experiment is as follows:

* 1. Launch the program
  2. Copy test data into the console
  3. Compare the result with the final result
  4. Analyze data

All data sets will be obtained from:

http://elib.zib.de/pub/mp-testdata/tsp/tsplib/atsp/index.html

# Data processing

The results for the experiment are as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Data set name | Dimensions | Heuristic value | Best known Solution[[9]](#footnote-9) | Percent difference |
| br17 | 17 | 39 | 39 | 0.00% |
| ft53 | 53 | 7449 | 6905 | 7.88% |
| ft70 | 70 | 39901 | 38673 | 3.18% |
| ftv33 | 34 | 1373 | 1286 | 6.77% |
| ftv35 | 35 | 1548 | 1473 | 5.09% |
| ftv38 | 39 | 1665 | 1530 | 8.82% |
| ftv44 | 45 | 1697 | 1613 | 5.21% |
| ftv47 | 48 | 1862 | 1776 | 4.84% |
| ftv55 | 56 | 1738 | 1608 | 8.08% |
| ftv64 | 65 | 2029 | 1839 | 10.33% |
| ftv70 | 71 | 2150 | 1950 | 10.26% |
| kro124 | 100 | 38935 | 36230 | 7.47% |
| p43 | 43 | 5625 | 5620 | 0.09% |
| ry48p | 48 | 14891 | 14422 | 3.25% |

# Results discussion and conclusion

The algorithm has given very good results compared to the respective best-known solutions, with none of the data sets exceeding an 11% difference from the best-known solution and has run for a reasonable amount of time.

Something that I noticed when my algorithm returned results was that it had many overlapping results. This has greatly reduced its ability to explore more possible solutions.

If we were to only look at accuracy, I would consider the polynomial expansion algorithm a suitable heuristic algorithm for solving the asymmetric travelling salesman problem.

# Appendix

The following is the code that I have used in the essay.

## A1: deltaMatrix.h

#pragma once

#include "matrix.h"

#include "infDouble.h"

#include <vector>

inline std::ostream& operator<< (std::ostream& output, std::vector<std::pair<infDouble, int>>& target) {

std::cout << "{ ";

for (std::pair<infDouble, int>& temp : target)

{

std::cout << temp.second << ' ';

}

std::cout << '}';

return output;

}

class deltaMatrix

{

matrix<infDouble>& \_graph;

void deriveDelta(int x, int y);

void deriveAllDelta();

public:

matrix<std::vector<std::pair<infDouble, int>>>\* body\_deltaMatrix;

deltaMatrix(matrix<infDouble>& in\_graph);

};

## A2: deltaMatrix.cpp

#include "matrix.h"

#include "infDouble.h"

#include <algorithm>

#include <vector>

bool sortCriteria(std::pair<infDouble, int> term1, std::pair<infDouble, int> term2)

{

return term1.first < term2.first;

}

//takes in a matrix and does preprocessing procedures to it

class deltaMatrix

{

matrix<infDouble>& \_graph;

public:

matrix<std::vector<std::pair<infDouble, int>>>\* body\_deltaMatrix = NULL;

private:

//calculates all the change values from node x to node y

void deriveDelta(int x, int y) {

if (x == y) {

body\_deltaMatrix->body\_matrix[x][y].push\_back(std::make\_pair(infDouble::OMEGA, -1));

return;

}

static int a;

static infDouble length;

length = \_graph.body\_matrix[x][y];

for (a = 0; a < body\_deltaMatrix->getM\_horizontalLength(); a++) {

if (a == x || a == y) { continue; }

body\_deltaMatrix->body\_matrix[x][y].push\_back({ \_graph.body\_matrix[x][a] + \_graph.body\_matrix[a][y] - length, a });

}

std::sort(body\_deltaMatrix->body\_matrix[x][y].begin(), body\_deltaMatrix->body\_matrix[x][y].end(), sortCriteria);

}

void deriveAllDelta() {

int y;

infDouble minVal, temp;

for (int x = 0; x < \_graph.getM\_horizontalLength(); x++)

{

for (y = 0; y < \_graph.getN\_verticalLength(); y++)

{

deriveDelta(x, y);

}

}

}

public:

deltaMatrix(matrix<infDouble>& in\_graph);

};

deltaMatrix::deltaMatrix(matrix<infDouble>& in\_graph) : \_graph(in\_graph) {

body\_deltaMatrix = new matrix<std::vector<std::pair<infDouble, int>>>(\_graph.getM\_horizontalLength(), \_graph.getN\_verticalLength());

deriveAllDelta();

}

## A3: polygon.h

#pragma once

#include <list>

#include <set>

#include <algorithm>

#include <vector>

#include "infDouble.h"

#include "deltaMatrix.h"

class polygonSet {

bool activated = false;

matrix<infDouble>& \_graph;

deltaMatrix\* \_delta;

int \_size = 0;

public:

polygonSet(matrix<infDouble>& in\_graph);

deltaMatrix& delta();

matrix<infDouble>& graph();

int size();

};

class polygon

{

std::list<int> body\_polygon;

public:

polygonSet& parent;

private:

infDouble \_length = 0.0;

int size = 0;

std::set<int> coveredPoints;

public:

polygon(polygonSet& in\_parent, int vertex1, int vertex2);

//pushes the node with the lowest delta value into the polygon

void advance();

polygon& minTSPCycle();

infDouble length() const;

};

## A4: deltaMatrix.cpp

#include "matrix.h"

#include "infDouble.h"

#include <algorithm>

#include <vector>

bool sortCriteria(std::pair<infDouble, int> term1, std::pair<infDouble, int> term2)

{

return term1.first < term2.first;

}

//takes in a matrix and does preprocessing procedures to it

class deltaMatrix

{

matrix<infDouble>& \_graph;

public:

matrix<std::vector<std::pair<infDouble, int>>>\* body\_deltaMatrix = NULL;

private:

//calculates all the change values from node x to node y

void deriveDelta(int x, int y) {

if (x == y) {

body\_deltaMatrix->body\_matrix[x][y].push\_back(std::make\_pair(infDouble::OMEGA, -1));

return;

}

static int a;

static infDouble length;

length = \_graph.body\_matrix[x][y];

for (a = 0; a < body\_deltaMatrix->getM\_horizontalLength(); a++) {

if (a == x || a == y) { continue; }

body\_deltaMatrix->body\_matrix[x][y].push\_back({ \_graph.body\_matrix[x][a] + \_graph.body\_matrix[a][y] - length, a });

}

std::sort(body\_deltaMatrix->body\_matrix[x][y].begin(), body\_deltaMatrix->body\_matrix[x][y].end(), sortCriteria);

}

void deriveAllDelta() {

int y;

infDouble minVal, temp;

for (int x = 0; x < \_graph.getM\_horizontalLength(); x++)

{

for (y = 0; y < \_graph.getN\_verticalLength(); y++)

{

deriveDelta(x, y);

}

}

}

public:

deltaMatrix(matrix<infDouble>& in\_graph);

};

deltaMatrix::deltaMatrix(matrix<infDouble>& in\_graph) : \_graph(in\_graph) {

body\_deltaMatrix = new matrix<std::vector<std::pair<infDouble, int>>>(\_graph.getM\_horizontalLength(), \_graph.getN\_verticalLength());

deriveAllDelta();

}

## A5: infDouble.h

#pragma once

#pragma once

#include <iostream>

#include <limits>

// double plus infinite portion

class infDouble

{

using idou = infDouble;

double normalValue = 0;

int infiniteValue = 0;

public:

infDouble();

infDouble(double x);

infDouble(double x, int y);

infDouble(std::string input);

infDouble(char input);

static idou OMEGA;

friend std::istream& operator>> (std::istream& input, infDouble& target);

friend std::ostream& operator<< (std::ostream& output, infDouble& target);

// operator overloading stuffs

bool operator> (idou other) const;

bool operator> (std::string other) const;

bool operator>= (idou other) const;

bool operator>= (std::string other) const;

bool operator< (idou other) const;

bool operator< (std::string other) const;

bool operator<= (idou other) const;

bool operator<= (std::string other) const;

bool operator== (idou other) const;

bool operator== (std::string other) const;

bool operator!= (idou other) const;

bool operator!= (std::string other) const;

idou operator+ (idou other) const;

idou operator+ (std::string other) const;

idou& operator+=(idou other);

idou& operator+= (std::string other);

idou& operator++();

idou operator- (idou other) const;

idou operator- (std::string other) const;

idou& operator-=(idou other);

idou& operator-= (std::string other);

idou& operator--();

operator double() const;

};

std::istream& operator>> (std::istream& input, infDouble& target);

std::ostream& operator<< (std::ostream& output, infDouble& target);

## A6: infDouble.cpp

#pragma once

#include <iostream>

#include <limits>

// double plus infinite portion

class infDouble {

using idou = infDouble;

double normalValue = 0;

int infiniteValue = 0;

public:

infDouble();

infDouble(double x);

infDouble(double x, int y);

infDouble(std::string input);

infDouble(char input);

static idou OMEGA;

friend std::istream& operator>> (std::istream& input, infDouble& target);

friend std::ostream& operator<< (std::ostream& output, infDouble& target);

// operator overloading stuffs

bool operator> (idou other) const;

bool operator> (std::string other) const;

bool operator>= (idou other) const;

bool operator>= (std::string other) const;

bool operator< (idou other) const;

bool operator< (std::string other) const;

bool operator<= (idou other) const;

bool operator<= (std::string other) const;

bool operator== (idou other) const;

bool operator== (std::string other) const;

bool operator!= (idou other) const;

bool operator!= (std::string other) const;

idou operator+ (idou other) const;

idou operator+ (std::string other) const;

idou& operator+=(idou other);

idou& operator+= (std::string other);

idou& operator++();

idou operator- (idou other) const;

idou operator- (std::string other) const;

idou& operator-=(idou other);

idou& operator-= (std::string other);

idou& operator--();

operator double() const;

};

infDouble::infDouble() : normalValue(0), infiniteValue(0) {}

infDouble::infDouble(double x) : normalValue(x), infiniteValue(0) {}

infDouble::infDouble(double x, int y) : normalValue(x), infiniteValue(y) {}

infDouble::infDouble(std::string input) : normalValue(0), infiniteValue(1) {}

infDouble::infDouble(char input) : normalValue(0), infiniteValue(1) {}

bool infDouble::operator> (infDouble other) const {

if (infiniteValue == other.infiniteValue) {

return normalValue > other.normalValue;

}

return infiniteValue > other.infiniteValue;

}

bool infDouble::operator> (std::string other) const { return operator>((idou)other); }

bool infDouble::operator>= (infDouble other) const {

if (infiniteValue == other.infiniteValue) {

return normalValue >= other.normalValue;

}

return infiniteValue >= other.infiniteValue;

}

bool infDouble::operator>= (std::string other) const { return operator>=((idou)other); }

bool infDouble::operator< (infDouble other) const {

if (infiniteValue == other.infiniteValue) {

return normalValue < other.normalValue;

}

return infiniteValue < other.infiniteValue;

}

bool infDouble::operator< (std::string other) const { return operator<((idou)other); }

bool infDouble::operator<= (infDouble other) const {

if (infiniteValue == other.infiniteValue) {

return normalValue <= other.normalValue;

}

return infiniteValue <= other.infiniteValue;

}

bool infDouble::operator<= (std::string other) const { return operator<=((idou)other); }

bool infDouble::operator== (infDouble other) const {

if (normalValue == other.normalValue) {

return infiniteValue == other.infiniteValue;

}

return false;

}

bool infDouble::operator== (std::string other) const { return operator==((idou)other); }

bool infDouble::operator!= (infDouble other) const {

if (normalValue == other.normalValue) {

return infiniteValue != other.infiniteValue;

}

return true;

}

bool infDouble::operator!= (std::string other) const { return operator!=((idou)other); }

infDouble infDouble::operator+ (infDouble other) const { return idou(normalValue + other.normalValue, infiniteValue + other.infiniteValue); }

infDouble infDouble::operator+ (std::string other) const { return idou(normalValue, infiniteValue + 1); }

infDouble& infDouble::operator+=(infDouble other) {

normalValue += other.normalValue;

infiniteValue += other.infiniteValue;

return \*this;

}

infDouble& infDouble::operator+= (std::string other) { infiniteValue++; return \*this; }

infDouble& infDouble::operator++() { infiniteValue++; return \*this; }

infDouble infDouble::operator- (infDouble other) const { return idou(normalValue - other.normalValue, infiniteValue - other.infiniteValue); }

infDouble infDouble::operator- (std::string other) const { return idou(normalValue, infiniteValue - 1); }

infDouble& infDouble::operator-=(infDouble other) {

normalValue -= other.normalValue;

infiniteValue -= other.infiniteValue;

return \*this;

}

infDouble& infDouble::operator-= (std::string other) { infiniteValue--; return \*this; }

infDouble& infDouble::operator--() { infiniteValue--; return \*this; }

infDouble::operator double() const {

if (infiniteValue > 0) {

// std::cerr << "error: positive infinite return value\n";

return DBL\_MAX;

}

if (infiniteValue < 0) {

// std::cerr << "error: negative infinite return value\n";

return DBL\_MIN;

}

return normalValue;

}

std::istream& operator>> (std::istream& input, infDouble& target)

{

static int temp;

try

{

if (!(input >> temp))

{

std::cin.clear();

std::string garbage;

input >> garbage;

throw 1;

}

target.normalValue = temp;

}

catch (int)

{

target.infiniteValue = 1;

}

return input;

}

std::ostream& operator<< (std::ostream& output, infDouble& target) {

if (target.infiniteValue != 0)

{

output << "inf";

}

else

{

output << target.normalValue;

}

return output;

}

infDouble infDouble::OMEGA = infDouble(DBL\_MAX, INT\_MAX);

## A7: matrix.h

#pragma once

//#include <vector>

#include <iostream>

#include <utility>

#include <iomanip>

template <class matrixBodyType>

//origin point at top left, format in [from][to]

class matrix

{

int M\_horizontalLength;

int N\_verticalLength;

bool active = false;

void destroy()

{

if (!active)

{

return;

}

for (int x = 0; x < M\_horizontalLength; x++)

{

delete[] body\_matrix[x];

}

delete[] body\_matrix;

body\_matrix = NULL;

active = false;

}

void reconstruct(int m, int n)

{

destroy();

body\_matrix = new matrixBodyType \* [m];

for (int x = 0; x < m; x++)

{

body\_matrix[x] = new matrixBodyType[n];

}

active = true;

}

public:

matrixBodyType\*\* body\_matrix = NULL;

matrix() = delete;

matrix(int m) : M\_horizontalLength(m), N\_verticalLength(m) { reconstruct(m, m); }

matrix(int m, int n) : M\_horizontalLength(m), N\_verticalLength(n) { reconstruct(m, n); }

~matrix() { destroy(); }

void resize(int m, int n)

{

destroy();

M\_horizontalLength = m;

N\_verticalLength = n;

reconstruct(m, n);

}

void fullInput()

{

int y;

for (int x = 0; x < M\_horizontalLength; x++)

{

for (y = 0; y < N\_verticalLength; y++)

{

std::cin >> body\_matrix[x][y];

}

}

}

void fullOutput()

{

int y;

for (int x = 0; x < M\_horizontalLength; x++)

{

for (y = 0; y < N\_verticalLength; y++)

{

std::cout << body\_matrix[x][y] << " ";

}

std::cout << '\n';

}

}

const int getM\_horizontalLength() const

{

return M\_horizontalLength;

}

const int getN\_verticalLength() const

{

return N\_verticalLength;

}

};

## A8: launch.cpp

#include "polygon.h"

int main() {

int size;

std::cin >> size;

matrix<infDouble> base(size);

base.fullInput();

polygonSet test(base);

infDouble min = infDouble::OMEGA;

int minX = 0, minY = 0;

for (int x = 0; x < size; x++) {

for (int y = 0; y < size; y++) {

if (x == y) { continue; }

polygon\* temp = new polygon(test, x, y);

if (temp->minTSPCycle().length() < min) {

min = temp->minTSPCycle().length();

minX = x;

minY = y;

}

std::cout << temp->minTSPCycle().length() << " ";

}

std::cout << std::endl;

}

std::cout << "-------------------------------------\n";

std::cout << "the minimum value found is: " << min << '\n';

std::cout << "at: " << minX << ", " << minY;

return 0;

}

# Bibliography

“The Problem.” The Travelling Salesman Problem. The University of Waterloo. Accessed November 31, 2021. https://www.math.uwaterloo.ca/tsp/problem/index.html.

“History.” The Travelling Salesman Problem. The University of Waterloo. Accessed November 31, 2021. https://www.math.uwaterloo.ca/tsp/history/index.html.

“Applications.” The Travelling Salesman Problem. The University of Waterloo. Accessed November 31, 2021. https://www.math.uwaterloo.ca/tsp/apps/index.html.

“Vector.” Encyclopædia Britannica. Encyclopædia Britannica, inc. Accessed February 1, 2022. https://www.britannica.com/science/vector-mathematics.

“Vector Addition.” from Wolfram MathWorld. Accessed February 1, 2022. https://mathworld.wolfram.com/VectorAddition.html.

Asymmetric tsps. Accessed February 1, 2022. http://elib.zib.de/pub/mp-testdata/tsp/tsplib/atsp-sol.html.

MP-TESTDATA - the TSPLIB asymmetric traveling salesman problem instances. Accessed February 1, 2022. http://elib.zib.de/pub/mp-testdata/tsp/tsplib/atsp/index.html.

1. “History.” The Travelling Salesman Problem. The University of Waterloo. Accessed November 31, 2021. https://www.math.uwaterloo.ca/tsp/history/index.html. [↑](#footnote-ref-1)
2. “The Problem.” The Travelling Salesman Problem. The University of Waterloo. Accessed November 31, 2021. https://www.math.uwaterloo.ca/tsp/problem/index.html. [↑](#footnote-ref-2)
3. “Applications.” The Travelling Salesman Problem. The University of Waterloo. Accessed November 31, 2021. https://www.math.uwaterloo.ca/tsp/apps/index.html. [↑](#footnote-ref-3)
4. “The Problem.” The Travelling Salesman Problem. [↑](#footnote-ref-4)
5. “Travelling Salesman Problem: Solution & Software to Solve TSP.” OptimoRoute, October 2, 2021. https://optimoroute.com/travelling-salesman-problem/. [↑](#footnote-ref-5)
6. “Number of Tours.” The Travelling Salesman Problem. The University of Waterloo. Accessed November 31, 2021. https://www.math.uwaterloo.ca/tsp/problem/index.html. [↑](#footnote-ref-6)
7. “Vector.” Encyclopædia Britannica. Encyclopædia Britannica, inc. Accessed February 1, 2022. https://www.britannica.com/science/vector-mathematics. [↑](#footnote-ref-7)
8. “Vector Addition.” from Wolfram MathWorld. Accessed February 1, 2022. https://mathworld.wolfram.com/VectorAddition.html. [↑](#footnote-ref-8)
9. Asymmetric tsps. Accessed February 1, 2022. http://elib.zib.de/pub/mp-testdata/tsp/tsplib/atsp-sol.html. [↑](#footnote-ref-9)