Assignment 1: Doubly Connected Edge List

Purpose

The purpose of this assignment is for you to:

- Improve your proficiency in C programming and your dexterity with dynamic memory allocation.
- Demonstrate understanding of a concrete data structure (doubly connected edge list).
- Practice multi-file programming and improve your proficiency in using UNIX utilities.

Problem Context

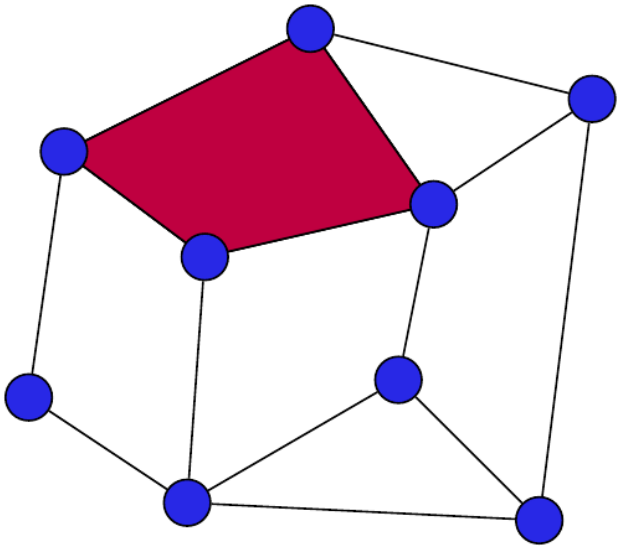
The context of the assignment task is that a set of watchtowers have been provided for the citizens of Victoria to protect against bushfires, funding has been allocated based on a number of regions and the government wants to better understand the total population the watchtowers in each region are watching over.

Background

Geometric data can be stored in many ways, such as a list of points, the union of basic shapes such as circles, rectangles and triangles or the solution of a set of inequalities. The choice of data structure can greatly affect which operations are easy to perform on that geometry.

Some geometric problems require us to not only store the geometric properties such as the coordinates of points (or vertices), but also if two points are connected or if two edges bound the same region in a polygonal mesh. Such information is called topological information.

There are a number of data structures that can store geometric and topological information efficiently. In this assignment, we will discuss one of those: the doubly connected edge list (DCEL). The DCEL is a data structure that enables us to store and reconstruct any planar subdivision. Now, what is a planar subdivision? Here is a graphical example but we give a more formal introduction below in the section Graphs.



As the diagram shows we have vertices colored in blue and edges colored in black. If you pick a single vertex of the red polygon and traverse the edges incident to it on clockwise order, then you will return to the starting vertex. Such a planar subdivision has in general three entities: vertices, edges and faces (the polygon regions).

We could store each polygon in a singly connected list of vertices but that would be quite inconvenient if we wanted to answer the following queries efficiently:

- Which edges share the same given vertex?
- Which two faces share a particular edge?
- Which faces are adjacent to a given face (for example, which polygons are adjacent to the red polygon)?
- Which edges border a given face?

In addition to queries, our data structure also needs to support various operations such as splitting the red face (polygon) if a new edge is introduced that connects two vertices of the face that are not adjacent or splitting a half-edge into two at the mid-point of a given edge.

Graphs

A graph is a G is a pair $G = \{V, E\}$ where V is a set of *nodes* or *vertices* and E is a set of edges (a binary relation on V).

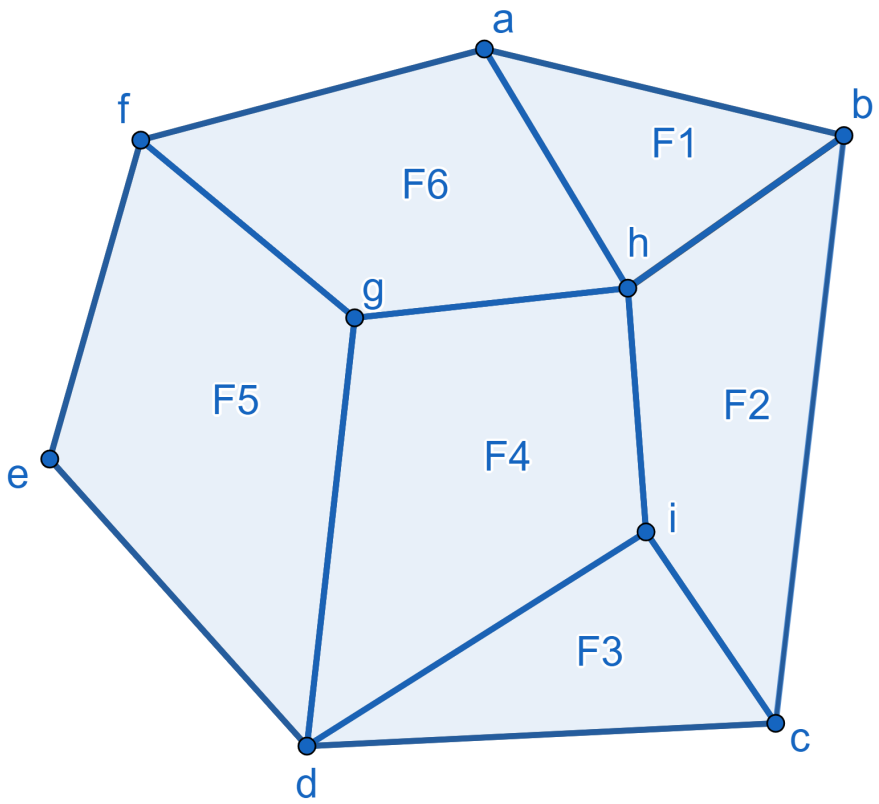
If a pair of vertices $(u, v) \in E$ is the same as the pair (v, u), they are connected by an *undirected* edge (u, v). The vertices are called *adjacent* to each other.

A graph is called *undirected* if all of its edges are undirected.

A graph is called *planar* if it is (a) an undirected graph and (b) can be drawn on a plane such that none of its edges are crossing.

A planar graph is a planar subdivision (see the figure above).

In this figure the vertices a and b are adjacent, and the edges (c, d), (d, i) and (i, c) describe the triangle (face) $F3$.

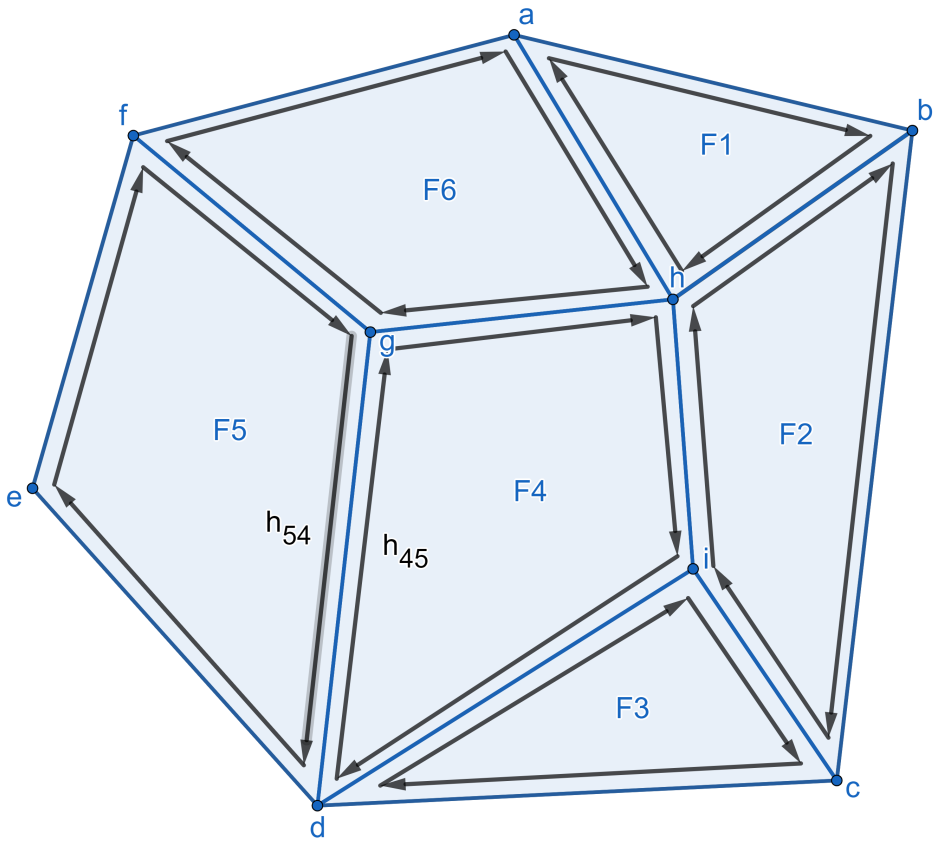


DCEL

If we were store the x and y-coordinates of the vertices a to i in an array or a linked list, then we

would lose all topological information, for example, that the points c, d and i form a triangle and all belong to the same face $F3$. And vice versa, we might be interested in listing all vertices in clock (or counter clock)-wise order of face $F5$. To capture this information and to answer these queries (efficiently), we need a DCEL.

A DCEL is a data structure that represents (stores) the vertices, edges, and faces of our planar graph (subdivision). The reason we do not simply use a doubly linked list, for example, to store a subdivision is that for a given edge e, there are two possible next edges because each edge is incident to two faces, for example, the edge (d, g) is incident to $F4$ and $F5$. In other words: there are two possible next edges. The key idea is to split each edge into two half edges, one where traverse a face in clockwise direction and one if we traverse the face in counter clockwise direction. Once we agree on an orientation (clockwise or counter clockwise), each face is bounded by unique half edges, i.e., each half edge is incident to exactly one face. The half edges that are incident with the same undirected edge are sometimes called *twins*. The figure below shows the twins h_{45} and h_{54} for the undirected edge (d, g). If we agree on clockwise orientation then the half edge h_{45} belongs exclusively to face $F4$ and h_{54} to $F5$.



For all details of the DCEL, head over to **Implementation Details**.

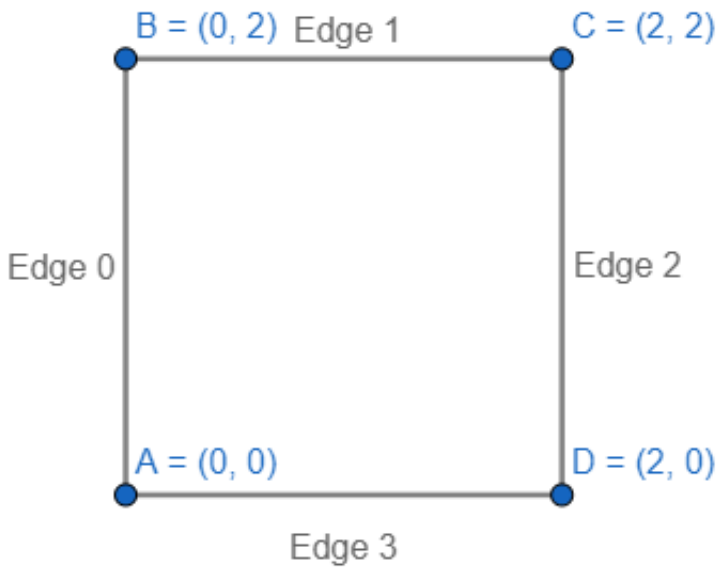
Your Task

In this assignment, you will create a fundamental data structure which supports the sorting of points of interest into partitions of a geometric space. These data come from properties of Victorian postcodes and a simple initial polygon. A user will be able to provide modifications to this polygon and receive information about what points land in each location.

Your implementation will receive three filenames as arguments and will build this data structure by reading from the first two files. The first file will contain a list of comma-separated values representing the location of a set of watch-towers and associated data for each of these watch towers (such as the population served) which you will read into a dynamically allocated array of pointers to a simple data structure containing this information.

The second file will contain a list of points, one per line, of the initial polygon, each of the coordinates separated by space. You may assume this polygon is always convex.

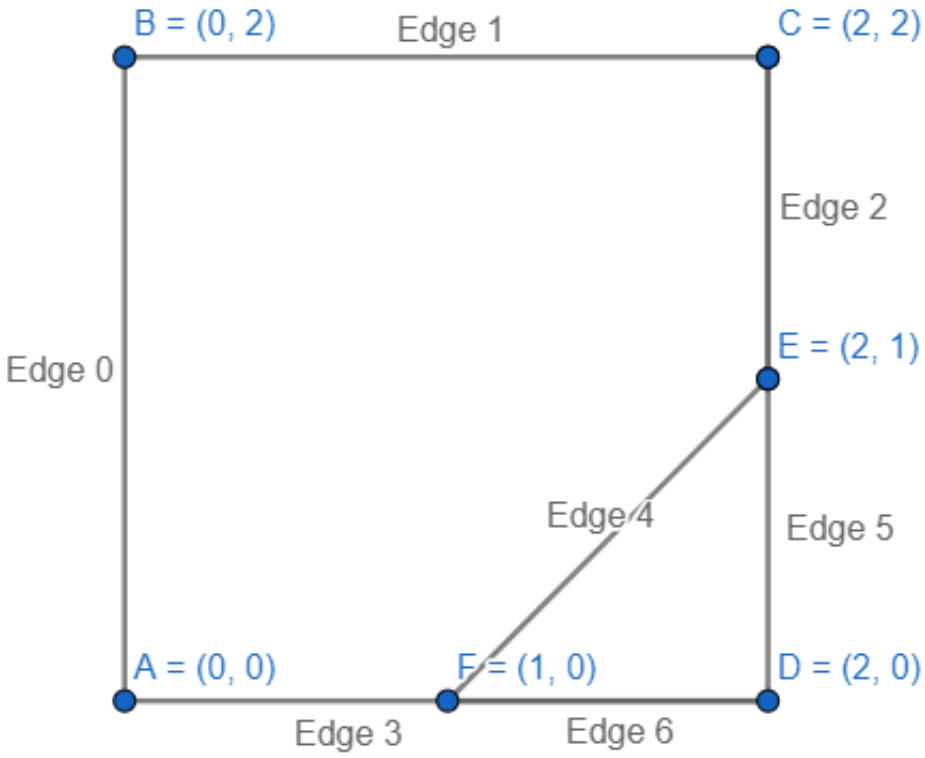
e.g. if the original polygon was the list of points [0,0], [0,2], [2,2], [2,0], this would generate 4 edges, [0,0] → [0,2] (edge 0), [0,2] → [2,2] (edge 1), [2,2] → [2,0] (edge 2) and [2,0] → [0,0] (edge 3).



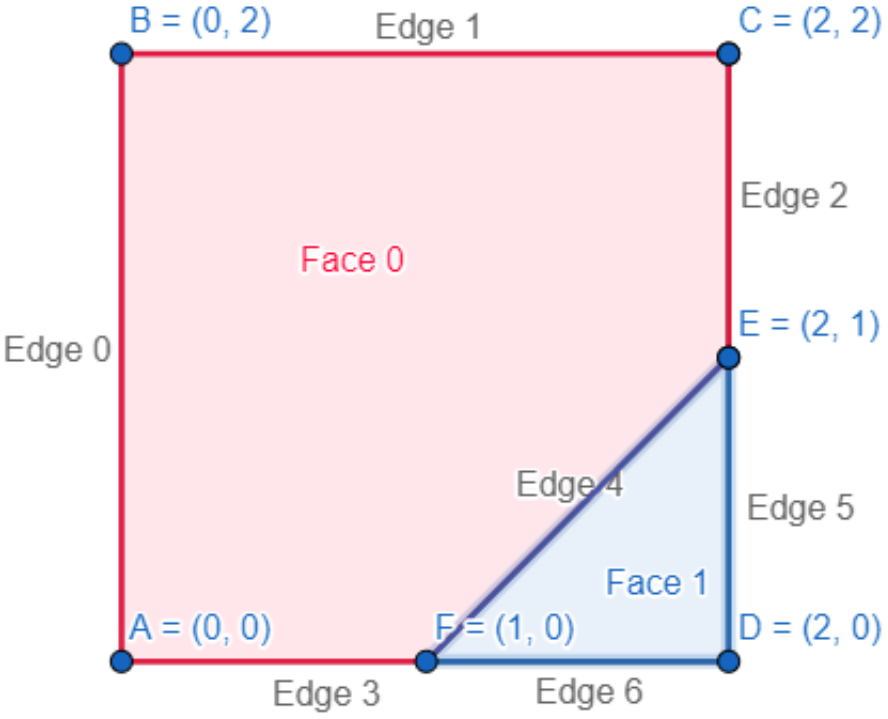
On standard input, you will receive a list of pairs of integers. Each pair represents a split which should occur in the polygon this split should bisect the two edges, beginning in the middle of the first edge and finishing at the middle of the second edge.

E.g. for the polygon above, the input 2, 3 would generate two additional points. The first point bisects edge 2 at position (2, 1) and the second points bisects edge 3 at position (0, 1). Those two new points are connected with an additional edge. This new edge is also numbered sequentially as Edge 4. The starting and ending edge are now split and 2 new edges are added (edge 5 and edge 6). Further details on the numbering of these splits are given in the Implementation Details section, but the

result of this is shown here:



This generates two areas, shown here as Face 0 and Face 1



The third file will be used to output the result of applying all splits. After all the splits have been processed, you'll output the information, first with the face, followed by each point of interest which lies inside that face. Finally you'll output the total population in each face.

Datasets

Watchtower Information

The watchtower data filename will be the first argument to your program. A sample dataset can be downloaded from the Dataset Download slide.

The postcode data initially comes from the Department of Environment, Land, Water and Planning (ANZVI0803003025), and has been combined with population data from the 2016 Census https://datapacks.censusdata.abs.gov.au/datapacks/ and names generated using the python package names (https://pypi.org/project/names/), which uses 1990 US Census data.

The synthetic dataset has the following fields

```
Watchtower ID - the ID of the watchtower (string)
Postcode - the postcode the watchtower is in (string)
Population Served - the population served by the watchtower in 2016 (integer)
Watchtower Point of Contact Name - the name of the manager of the watchtower (string)
x - the longitude of the watchtower location (double)
y - the latitude of the watchtower location (double)
```

The fields <Watchtower ID>, <Postcode> and <Watchtower Point of Contact Name> are alphabetic strings of varying length. You may assume none of these fields are more than 128 characters. <Population Served> is an integer with no characters separating thousands. <x> and <y> can be considered double precision numbers.

This data is in csv format, with each field separated by a comma. For the purposes of the assignment, you may assume commas never occur inside any of the fields, the input data are well-formatted, the input files are not empty, the input file includes a header, the fields always occur in the order given above and that the maximum length of an input record (a single full line of the csv) is 512 characters. This could help you choose a reading buffer size.

You should store this data in a dynamic array of pointers which can be accessed by the index of the array.

Smaller samples of various sizes are also provided.

Initial Polygon

The initial polygon filename will be the second argument to your program. Two sample polygons have been provided for you on the Dataset Download slide.

One is a square polygon and the other is an irregular polygon.

```
x - The longitude of the location of the polygon point (double)
```

```
y - The latitude of the location of the polygon point (double)
```

The fields <x> and <y> are decimals which should be interpreted with double precision. The data has no header line. The point in each line should be interpreted as being connected to the point in its following line, with the final line being connected to the first line.

The two values are split by a space.

Sample Splits

Splits should be read into your program from standard input. Two sample sets of splits for the square polygon and five sample sets of splits for the irregular polygon have been provided on the Dataset Download slide.

```
startEdge - The beginning edge of the split (integer)
endEdge - The final edge of the split (integer)
```

The fields <startEdge> and <endEdge> are integers and correspond to the edges explained in the Implementation Details slide.

The two values are split by a space and pairs are given one per line.

Implementation Details

Your `Makefile` should produce an executable program called `voronoi1`. This program should take three command line arguments: (1) the name of the csv data file used to construct the point data, (2) the second file will contain the initial polygon, specifying the x, y vertices in order, separated by spaces and (3) the name of an output file.

Your `voronoi1` program should:

- Read the data from the data file specified in the command line argument. The data from the csv should be stored in a dynamic array of pointers to structs. Datatypes for each field should be consistent with those in the Dataset section.
- Construct the initial polygon from the second file. The points from this file should be used to construct a single Doubly Connected Edge List representing the inside of this polygon, with the first edge in this Doubly Connected Edge List stored in a dynamic array.
- Split the polygon into additional faces based on a listing of pairs of edges, creating a new edge between the middle of the first edge in the pair and the middle of the second edge in the pair, this numbering should be consistent with the numbering explained in the *Numbering* subheading. The split will create exactly one new face.
- Output a summary of the watchtowers which lie in each face, consistent with the format explained under the *Format* subheading.

Doubly Connected Edge List

In a doubly connected edge list, each edge is represented by two "half-edges", for the purposes of this assignment each half-edge should have:

- The index of the vertex at the end of the half-edge.
- The index of the vertex at the start of the half-edge.
- A pointer to the next half-edge in the face.
- A pointer to the previous half-edge in the face.
- A pointer to the other half-edge (this other edge runs in the opposite direction).
- The index of its corresponding face.
- The index of the edge it forms a part of.

Each vertex should have:

- An x-position.
- A y-position.

Each edge should have:

- A pointer to either (non-NULL) half-edge in the edge.

Each face should have:

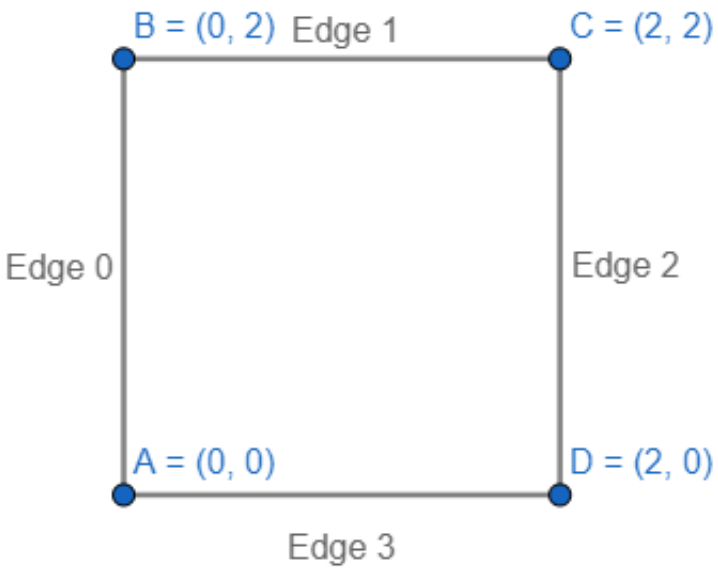
- A pointer to any half-edge in the face.

This choice of organisation is not always strict (e.g. a vertex might be stored inside each half-edge itself if you like), but is recommended as it will likely help you when debugging your program. In addition it is recommended that you use dynamic arrays to store your vertices, edges and faces, this will make freeing all your data a simple for loop.

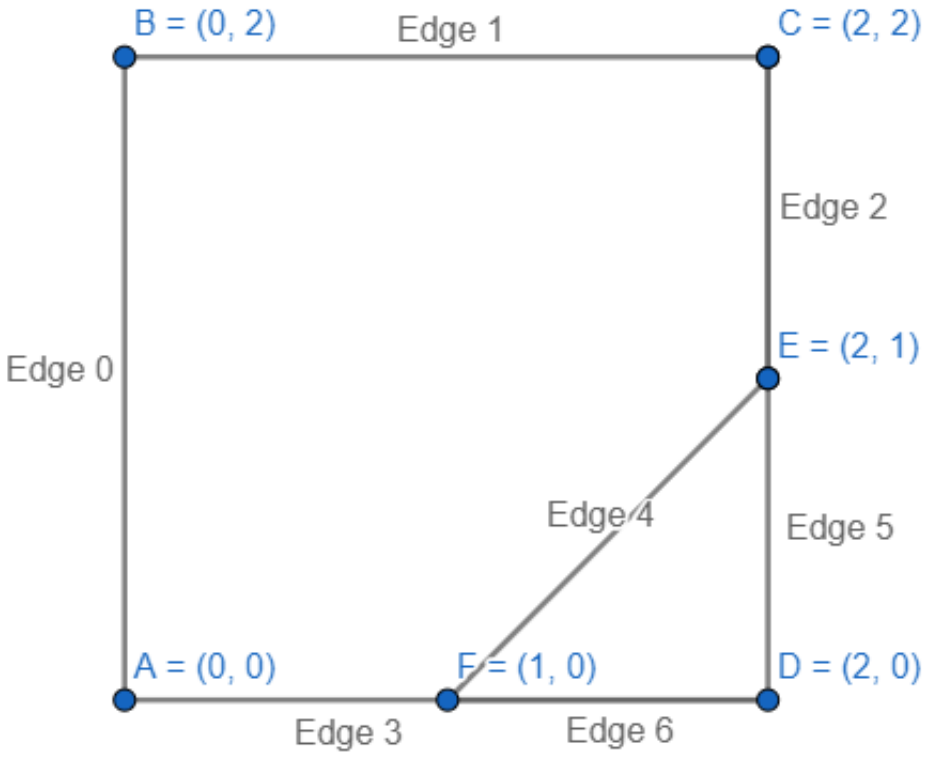
Numbering

We now revisit our earlier example of splitting a simple square face constructed with the points [0,0], [0,2], [2,2] and [2,0], which created four edges:

- [0,0] → [0,2] (edge 0),
- [0,2] → [2,2] (edge 1),
- [2,2] → [2,0] (edge 2) and
- [2,0] → [0,0] (edge 3)



In the split we add a split from the middle of Edge 2 to the middle of Edge 3. This operation is a little more complex than it first appears:



The construction of [2, 1] → [1, 0] (Edge 4) here is quite straight-forward, but as part of this operation we construct up to six half-edges and two vertices in total:

- [2, 1] (Vertex E)
- [1, 0] (Vertex F)
- [2, 1] → [1, 0] (Edge 4),
- Its pair, [1, 0] → [2, 1] (Edge 4),
- The edge from Edge 2's new *end* vertex to its old *end* vertex (Edge 5),
- If the half-edge for Edge 2 existed, we would also construct it,
- The edge from Edge 3's old *start* vertex to its new *start* vertex (Edge 6),
- If the half-edge for Edge 3 existed, we would also construct it.

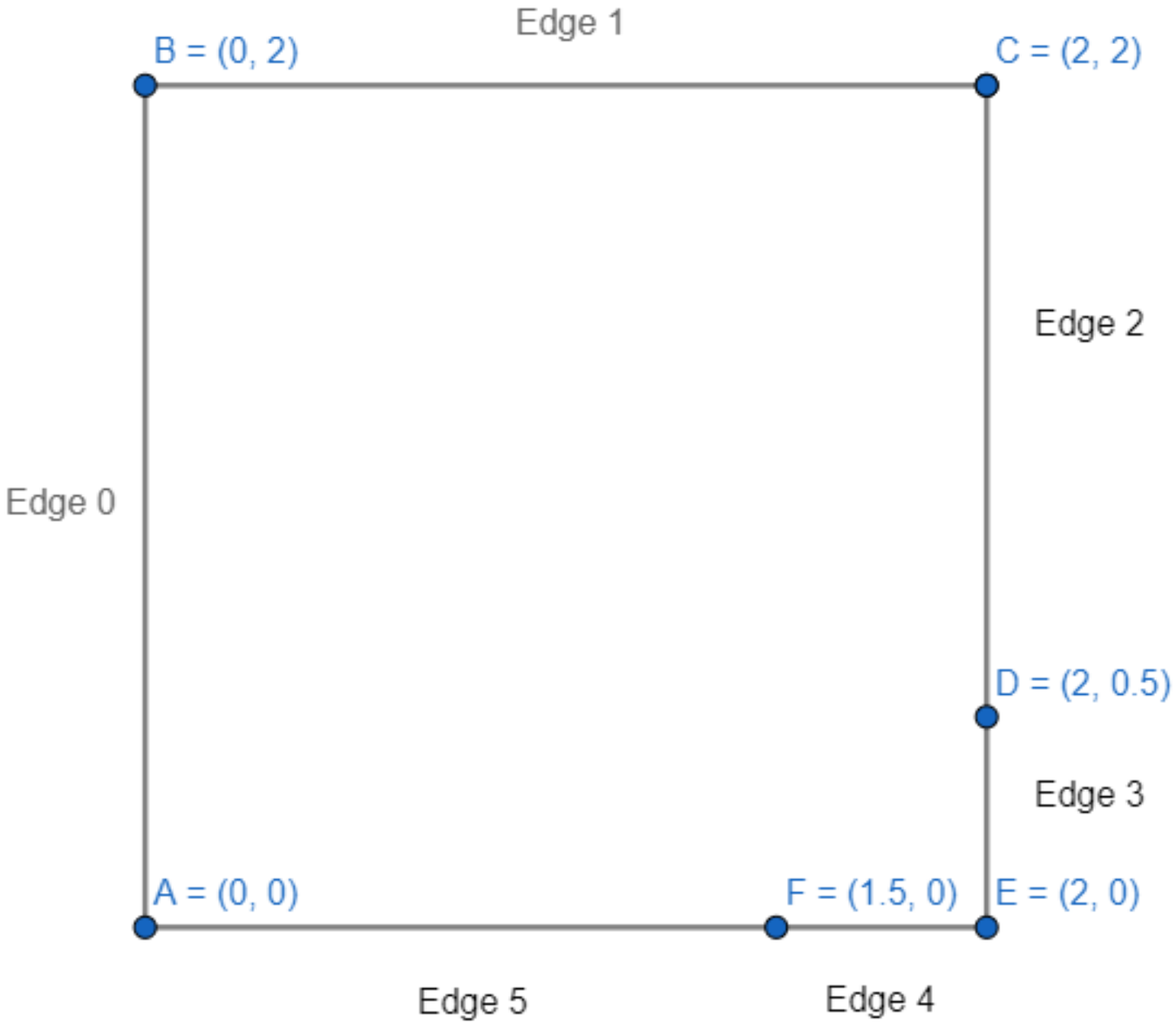
You may assume in this task that splits never need to cross one another (e.g. a line reading `5 0` would not be possible), this means every split creates exactly one additional face. Every half-edge in the new face must be updated to reflect its new face.

Non-Adjacent Splits

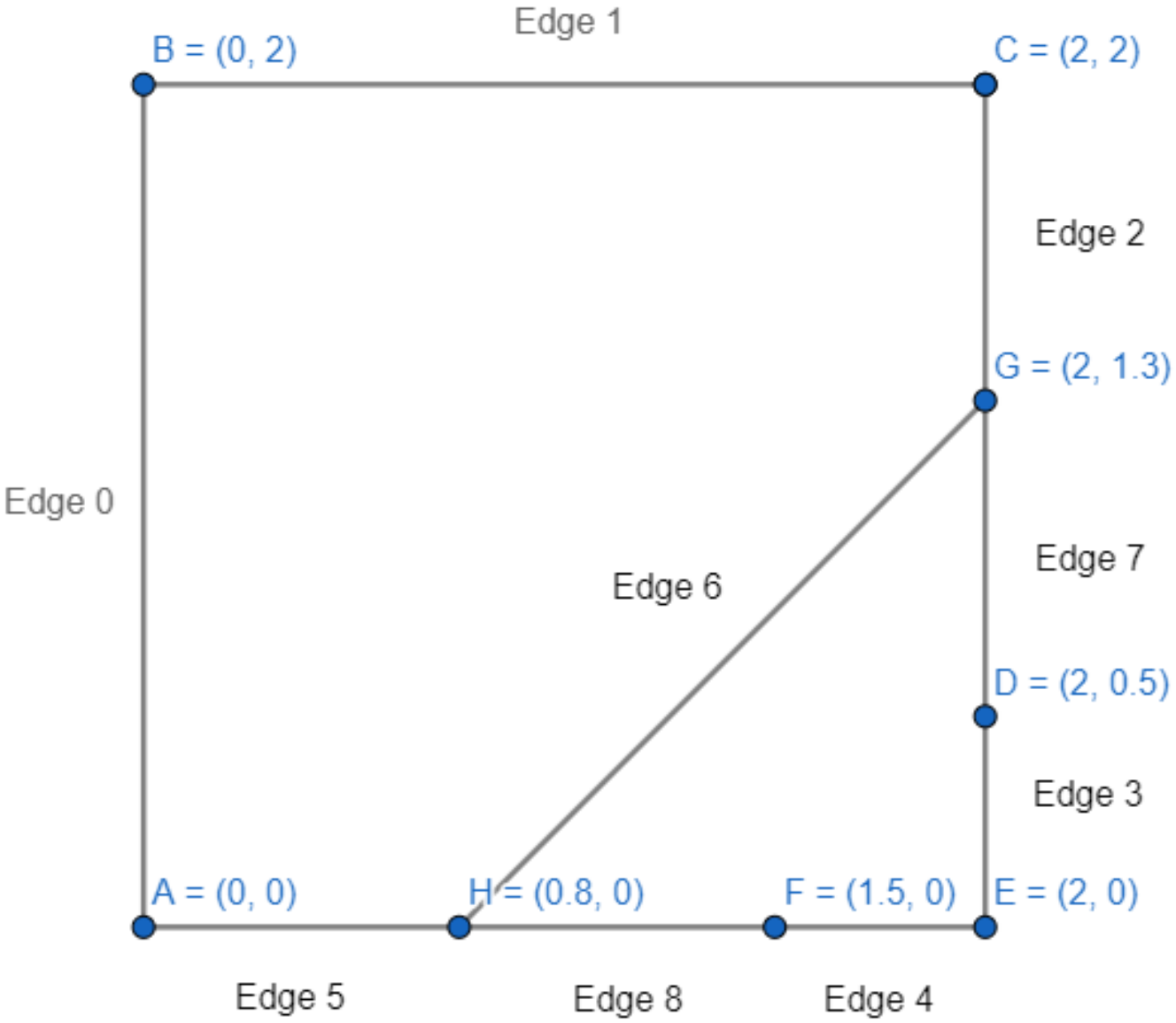
One additional complexity exists in performing a split. This is where the two edges are not adjacent. Consider this example given as six points:

```
0 0
0 2
2 2
2 0.5
2 0
```

```
1.5 0
```



If we now introduce a split from Edge 2 to Edge 5, edges are added in a similar way:



Note however that because the edges 2 and 5 are not *adjacent originally* (the next edge of edge 2 was not edge 5 *nor* was edge 5 the previous edge of edge 2), this means there is at least one edge between them, in this case:

1. The new half-edge of Edge 7 must be connected with the half-edge following Edge 2 (next), which is in Edge 3 in this case.
2. The old half-edge following Edge 2 (in Edge 3) must be connected to the new half-edge of Edge 7 (prev).
3. The old half-edge preceding Edge 5 (in Edge 4) must be connected to the new half-edge of Edge 8 (next).
4. The new half-edge of Edge 8 must be connected with the edge preceding Edge 5 (prev), which is in Edge 4 in this case.

For the destination of the split, only the immediately previous half-edge matters, and for the source of the split, only the immediately following half-edge matters. Note also that though this example is given had a distinct Edge 3 and Edge 4, if only one edge originally separated Edge 2 and Edge 5, both

new half-edges would connect to the same half-edge without any further issues.

Format

Your program should output, in increasing order, the faces. Firstly printing the face number (beginning from 0), then details of each watchtower in the face, with each field printed out along with its column headers, each field's value separated from its field name with a colon and space and each field separated by a comma, with one watchtower per line for each watchtower. Finally, after all faces and their watchtowers have been printed, output, for each face, the total population served by all the watchtowers contained in it. Assume where multiple watchtowers serve the same postcode that these populations are independent and can simply be summed.

Example

For testing, it may be convenient to create a file of keys to be searched, one per line, and redirect the input from this file. Use the UNIX operator < to redirect input from a file.

Examples of use:

```
make voronoi1
#./voronoi1 datafile polygonfile outputfile < splitsfile
./voronoi1 s2ds.csv victoria_square.txt output.txt < single_split.txt
```

Example output (outputfile):

```
0
Watchtower ID: WT3953SGAEI, Postcode: 3953, Population Served: 1571, Watchtower Point of Contact Na
1
Watchtower ID: WT3030WFXSP, Postcode: 3030, Population Served: 16718, Watchtower Point of Contact N
Watchtower ID: WT3030EAAIV, Postcode: 3030, Population Served: 16718, Watchtower Point of Contact N
Face 0 population served: 1571
Face 1 population served: 33436
```

There may be variations in the numerical precision in the output and dataset characteristics may not be reflected in any of the provided datasets.

Requirements

The following implementation requirements must be adhered to:

- You must write your implementation in the C programming language.
- You must write your code in a modular way, so that your implementation could be used in another program without extensive rewriting or copying. This means that the Doubly Connected Edge List operations are kept together in a separate .c file, with its own header (.h) file, separate from the main program.
- Your code should be easily extensible to different doubly connected edge lists. This means that the functions for modifying parts of your doubly connected edge list should take as arguments not only the values required to perform the operation required, but also a pointer to a particular doubly connected edge list, e.g. newSplit(dcel, sourceEdge, destinationEdge).
- Your implementation must read the input file once only.
- Your program should store strings in a space-efficient manner. If you are using malloc() to create the space for a string, remember to allow space for the final end of string '\0' (NULL).
- A full Makefile is not provided for you. The Makefile should direct the compilation of your program. To use the Makefile, make sure it is in the same directory as your code, and type `make voronoi1` to make the dictionary. You must submit your makefile with your assignment.

Hint: If you haven't used make before, try it on simple programs first. If it doesn't work, read the error messages carefully. A common problem in compiling multifile executables is in the included header files. Note also that the whitespace before the command is a tab, and not multiple spaces. It is not a good idea to code your program as a single file and then try to break it down into multiple files. Start by using multiple files, with minimal content, and make sure they are communicating with each other before starting more serious coding.

Programming Style

Below is a style guide which assignments are evaluated against. For this subject, the 80 character limit is a guideline rather than a rule - if your code exceeds this limit, you should consider whether your code would be more readable if you instead rearranged it.

```
/** ***********************
* C Programming Style for Engineering Computation
* Created by Aidan Nagorcka-Smith (aidann@student.unimelb.edu.au) 13/03/2011
* Definitions and includes
* Definitions are in UPPER_CASE
* Includes go before definitions
* Space between includes, definitions and the main function.
* Use definitions for any constants in your program, do not just write them
* in.
*
* Tabs may be set to 4-spaces or 8-spaces, depending on your editor. The code
* Below is ``gnu'' style. If your editor has ``bsd'' it will follow the 8-space
* style. Both are very standard.
*/

/**
* GOOD:
*/

#include <stdio.h>
#include <stdlib.h>
#define MAX_STRING_SIZE 1000
#define DEBUG 0
int main(int argc, char **argv) {
 ...

/**
* BAD:
*/

/* Definitions and includes are mixed up */
#include <stdlib.h>
#define MAX_STING_SIZE 1000
/* Definitions are given names like variables */
#define debug 0
#include <stdio.h>
/* No spacing between includes, definitions and main function*/
int main(int argc, char **argv) {
 ...

/** ****************************
* Variables
* Give them useful lower_case names or camelCase. Either is fine,
```

```
* as long as you are consistent and apply always the same style.
* Initialise them to something that makes sense.
*/

/**
* GOOD: lower_case
*/

int main(int argc, char **argv) {

  int i = 0;
  int num_fifties = 0;
  int num_twenties = 0;
  int num_tens = 0;

 ...
/**
* GOOD: camelCase
*/

int main(int argc, char **argv) {

  int i = 0;
  int numFifties = 0;
  int numTwenties = 0;
  int numTens = 0;

 ...
/**
* BAD:
*/

int main(int argc, char **argv) {

  /* Variable not initialised - causes a bug because we didn't remember to
  * set it before the loop */
  int i;
  /* Variable in all caps - we'll get confused between this and constants
  */
  int NUM_FIFTIES = 0;
  /* Overly abbreviated variable names make things hard. */
  int nt = 0

  while (i < 10) {
    ...
    i++;
  }

 ...

/** ********************
* Spacing:
* Space intelligently, vertically to group blocks of code that are doing a
* specific operation, or to separate variable declarations from other code.
```

```
* One tab of indentation within either a function or a loop.
* Spaces after commas.
* Space between ) and {.
* No space between the ** and the argv in the definition of the main
* function.
* When declaring a pointer variable or argument, you may place the asterisk
* adjacent to either the type or to the variable name.
* Lines at most 80 characters long.
* Closing brace goes on its own line
*/

/**
* GOOD:
*/

int main(int argc, char **argv) {

 int i = 0;

 for(i = 100; i >= 0; i--) {
   if (i > 0) {
     printf("%d bottles of beer, take one down and pass it around,"
     " %d bottles of beer.\n", i, i - 1);
   } else {
     printf("%d bottles of beer, take one down and pass it around."
     " We're empty.\n", i);
   }
 }

 return 0;
}

/**
* BAD:
*/

/* No space after commas
* Space between the ** and argv in the main function definition
* No space between the ) and { at the start of a function */
int main(int argc,char ** argv){
 int i = 0;
 /* No space between variable declarations and the rest of the function.
 * No spaces around the boolean operators */
 for(i=100;i>=0;i--) {
 /* No indentation */
 if (i > 0) {
 /* Line too long */
 printf("%d bottles of beer, take one down and pass it around, %d
bottles of beer.\n", i, i - 1);
 } else {
 /* Spacing for no good reason. */

 printf("%d bottles of beer, take one down and pass it around."
 " We're empty.\n", i);
```

```
 }
 }
 /* Closing brace not on its own line */
 return 0;}

/** ****************
* Braces:
* Opening braces go on the same line as the loop or function name
* Closing braces go on their own line
* Closing braces go at the same indentation level as the thing they are
* closing
*/

/**
* GOOD:
*/

int main(int argc, char **argv) {

  ...

  for(...) {
    ...
  }

  return 0;
}

/**
* BAD:
*/

int main(int argc, char **argv) {

  ...

  /* Opening brace on a different line to the for loop open */
  for(...)
  {
    ...
    /* Closing brace at a different indentation to the thing it's
    closing
    */
           }

 /* Closing brace not on its own line. */
       return 0;}

/** **************
* Commenting:
* Each program should have a comment explaining what it does and who created
* it.
* Also comment how to run the program, including optional command line
```

```
* parameters.
* Any interesting code should have a comment to explain itself.
* We should not comment obvious things - write code that documents itself
*/

/**
* GOOD:
*/

/* change.c
*
* Created by Aidan Nagorcka-Smith (aidann@student.unimelb.edu.au)
13/03/2011
*
* Print the number of each coin that would be needed to make up some
change
* that is input by the user
*
* To run the program type:
* ./coins --num_coins 5 --shape_coins trapezoid --output blabla.txt
*
* To see all the input parameters, type:
* ./coins --help
* Options::
*   --help                Show help message
*   --num_coins arg       Input number of coins
*   --shape_coins arg     Input coins shape
*   --bound arg (=1)      Max bound on xxx, default value 1
*   --output arg          Output solution file
*
*/

int main(int argc, char **argv) {

  int input_change = 0;

  printf("Please input the value of the change (0-99 cents
  inclusive):\n");
  scanf("%d", &input_change);
  printf("\n");

  // Valid change values are 0-99 inclusive.
  if(input_change < 0 || input_change > 99) {
    printf("Input not in the range 0-99.\n")
  }

  ...

/**
* BAD:
*/

/* No explanation of what the program is doing */
int main(int argc, char **argv) {
```

```
  /* Commenting obvious things */
  /* Create a int variable called input_change to store the input from
  the
  * user. */
  int input_change;

  ...

/** ****************
* Code structure:
* Fail fast - input checks should happen first, then do the computation.
* Structure the code so that all error handling happens in an easy to read
* location
*/

/**
* GOOD:
*/
if (input_is_bad) {
  printf("Error: Input was not valid. Exiting.\n");
  exit(EXIT_FAILURE);
}

/* Do computations here */
...

/**
* BAD:
*/

if (input_is_good) {
  /* lots of computation here, pushing the else part off the screen.
  */
  ...
} else {
  fprintf(stderr, "Error: Input was not valid. Exiting.\n");
  exit(EXIT_FAILURE);
}
```

Some automatic evaluations of your code style may be performed where they are reliable. As determining whether these style-related issues are occurring sometimes involves non-trivial (and sometimes even undecidable) calculations, a simpler and more error-prone (but highly successful) solution is used. You may need to add a comment to identify these cases, so check any failing test outputs for instructions on how to resolve incorrectly flagged issues.

Additional Support

Your tutors will be available to help with your assignment during the scheduled workshop times. Questions related to the assignment may be posted on the Ed discussion forum, using the folder tag Assignments for new posts. You should feel free to answer other students' questions if you are confident of your skills.

A tutor will check the discussion forum regularly, and answer some questions, but be aware that for some questions you will just need to use your judgment and document your thinking. For example, a question like, "How much data should I use for the experiments?", will not be answered; you must try out different data and see what makes sense.

If you have questions about your code specifically which you feel would reveal too much of the assignment, feel free to post a private question on the discussion forum.

Submission

Your C code files (including your Makefile and any other files needed to run your code) should be submitted through Ed to this assignment. Your programs must compile and run correctly on Ed. You may have developed your program in another environment, but it still must run on Ed at submission time. For this reason, and because there are often small, but significant, differences between compilers, it is suggested that if you are working in a different environment, you upload and test your code on Ed at reasonably frequent intervals.

A common reason for programs not to compile is that a file has been inadvertently omitted from the submission. Please check your submission, and resubmit all files if necessary.

Assessment

There are a total of 10 marks given for this assignment.

Your C program will be marked on the basis of accuracy, readability, and good C programming structure, safety and style, including documentation (2 marks). Safety refers to checking whether opening a file returns something, whether mallocs do their job, etc. The documentation should explain all major design decisions, and should be formatted so that it does not interfere with reading the code. As much as possible, try to make your code self-documenting, by choosing descriptive variable names. The remainder of the marks will be based on the correct functioning of your submission.

Note that these correct functioning-related marks will be based on passing various tests. If your program passes these tests without addressing the learning outcomes (e.g. if you fully hard-code solutions or otherwise deliberately exploit the test cases), you may receive less marks than is suggested but your marks will otherwise be determined by test cases.

```
| Marks | Task
|-------|-------------------------------------------------------------------------------
| 3     | Able to read and output dataset (no splits) with no memory errors and all memory freed.
| 2     | Splits on square polygon work correctly with no memory errors and all memory freed.
| 3     | Splits on irregular polygon work correctly with no memory errors and all memory freed.
| 2     | Program style consistent with Programming Style slide. Memory allocations and file opens
```

Note that code style will be manually marked in order to provide you with the most meaningful feedback for the second assignment.

Hints

Here are a number of suggestions on how to approach the task in parts:

1. Begin with reading in the small file and storing it into an array of structs, this is generally non-trivial!
2. Get your program working cleanly on the smallest dataset in the first part before moving on.
3. After your program works on the smallest dataset, move up through the datasets until you correctly store all the datasets.
4. Once data is being correctly read, build the initial polygon and check your program is producing the correct output for this simple test, fully in the correct format.
5. After your program is fully working, read in the splits and start tackling the DCEL functionality.

A few general tips which are likely to apply when constructing the polygon and other geometry:

- Where possible, if pointers change meaning in the course of an operation, introduce additional variables to hold each value so you don't need to think about what variables in motion refer to.

Compare:

```
void swapVals(int *a, int *b){
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

with:

```
void swapVals(int *a, int *b){
    int oldA = *a;
    int oldB = *b;
    *a = oldB;
    *b = oldA;
}
```

The latter form is not vulnerable to the error:

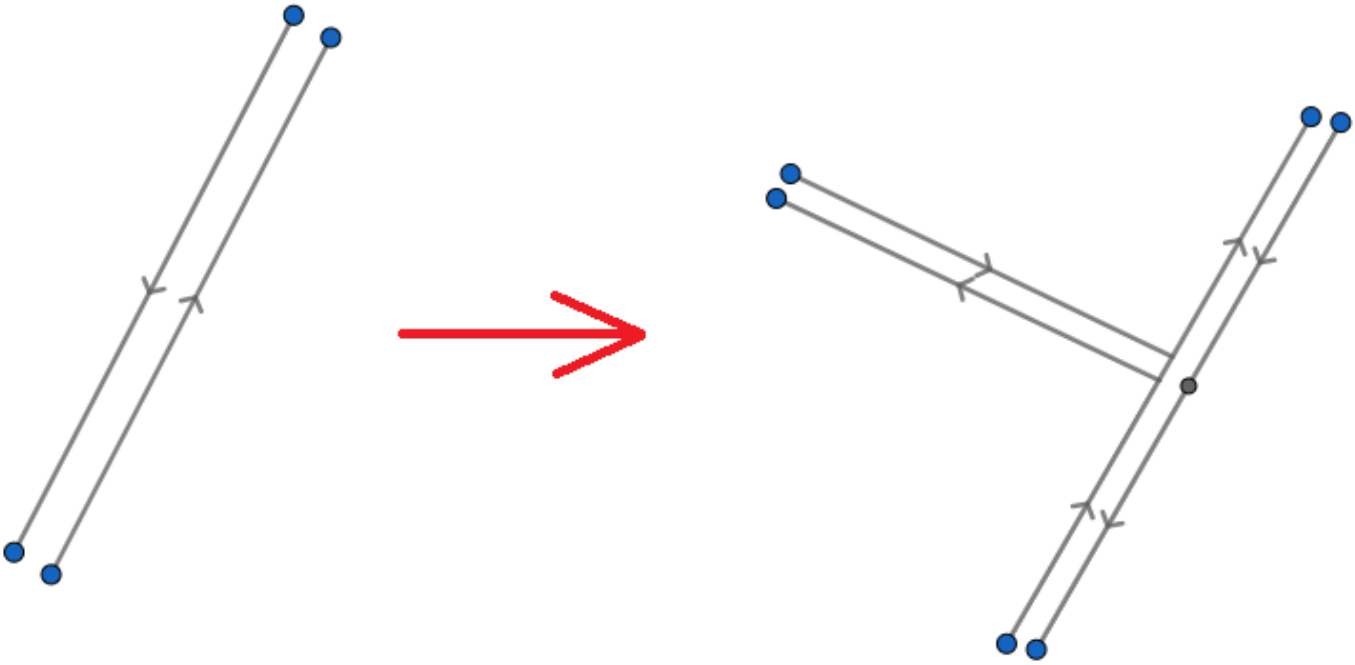
```
void swapVals(int *a, int *b){
    *a = *b; // *a = oldB;
    *b = *a; // *b = oldB; (!!)
}
```

The introduction of temporary variables which represent our conceptual understanding avoids mental load. The transformation from the second to the first is a trivial operation for the compiler to perform, so feel free to treat this as a free lunch and use as many temporary variables as you like!

- In a similar vein, split your program into distinct sections where possible - or, to put it another way, prepare as much as you can before you "begin". e.g. If, part-way through your calculation, you might need to reallocate something to get space for it, it is often possible to move this check *before* you begin the calculation. This allows you to think fully about the calculation.
- This assignment is very geometric, so draw diagrams! The split part of the task has a decent amount of code, but many of the steps can be completed mostly independently. Setting up vertices, updating pointers, creating all the halfEdges you need, etc. can all be ticked off one-by-one with a good diagram.

DCEL Splits with Pairs

The Doubly Connected Edge List assumes that each half-edge is connected to exactly one other half-edge, this implicitly means that a split of an edge on one side will create a half-edge on the other side, even though this split is not "necessary" to represent the geometry. See in this example how a split originating from the grey midpoint on the left half-edge also produces a new half-edge on the paired half-edge.



If an edge has no paired half-edge, we of course do not need to split its half-edge pair.

Note also that this new half-edge splits an edge which previously was associated with one face, which gives us a few simpler steps:

- The new paired halfEdge has the same face.
- Because the original halfEdge is split, you only need to connect the old halfEdge to the new halfEdge (the new edge will be inserted before or after in the ordering), the other side of the halfEdge will remain correct without modification.

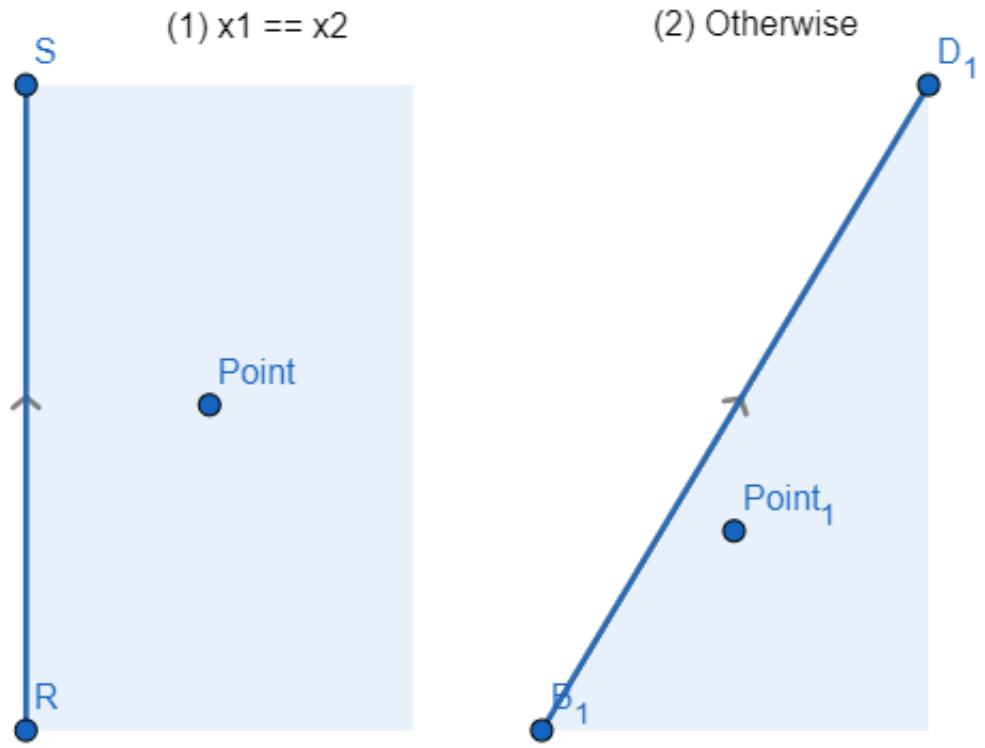
- You can then connect the remaining surrounding halfEdge to the new halfEdge.
    - If the new halfEdge is inserted before the old halfEdge, you'll need to connect the next pointer of the old previous halfEdge to the new halfEdge.
    - If the new halfEdge is inserted after the old halfEdge, you'll need to connect the prev pointer of the previous halfEdge to the new halfEdge.

Determining Whether a Given Point is in a Particular Face

To see if a point is in a particular face, we highlight the idea of a half-plane. A half-plane is essentially all the points on one side of a particular line. If a point is in the correct half-plane for all halfEdges in a particular face. Because you can assume all polygons constructed are convex, this check need only verify the point is on the *same side* of the half-plane for all halfEdges in the face. To check this, you can start at the first halfEdge in the face and traverse all halfEdges in the face until you return back to the first halfEdge.

Determining Which Side a Given Point is of a Half-Plane

There are two cases for half-plane checking which you might have to look at:



1. In the first case, the x-coordinate of both the start and end vertex are the same. Because we use what's known as a clockwise winding order, if the point we're checking has a greater x-coordinate then it will lie inside the shape. Otherwise it lies outside the point.
2. In the second case, we can use the simple y = mx + c formula.
    - First we calculate the gradient, m, and intercept, c

$$m = \frac{y_2 - y_1}{x_2 - x_1},\ c = y_2 - mx_2$$

- We then use this gradient and intercept to see what we'd expect the point's y-coordinate to be if it lay on the line between v_1 and v_2

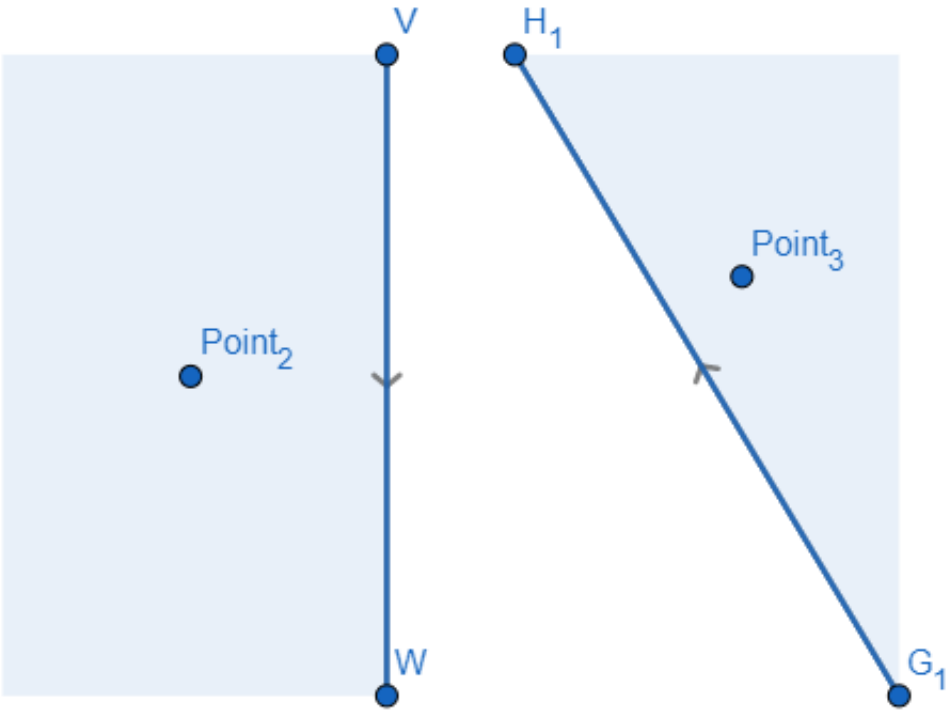
$$y_{predicted} = mx + c$$

- We then use this value to see where the y-coordinate is in relation to this value

$$y_r = y - y_{predicted}$$

- If $y_r \leq 0$, the point is inside the shape.

Note that the above only applies when the points forming the halfEdge are ordered by increasing x-coordinate (or increasing y-coordinate in the case of equal x-coordinates). When the order is x-coordinate is decreasing (or x-coordinate is equal and y-coordinate is decreasing), we look for the opposite relations



- When $x_1 == x_2$ and $y_1 \geq y_2$, the point is in the shape if $x \leq x_1$
- Otherwise, if $x_1 \geq x_2$ and $y_r \geq 0$, the point is in the shape.

Updating Changing Faces

Every time you add a split to your DCEL data structure, you will create exactly one extra face. You can start at the edge on the other side of the new edge comprising the split, updating every face until you return to the original edge.

However, there is one risk here! The face refers to one half-edge in the face, if we aren't careful, we might change the half-edge we've made the face refer to the new face - leading to two edges which

both refer to the same face. The solution here is simple, check if the face of the edge on the side you didn't update isn't still in the right face. If it isn't, update the face to point to a half-edge on the other side.

Debugging

Though debugging is very much a general skill for most of the task, debugging geometric or structured information can present an additional challenge! We will provide some tools you can use to visualise and investigate the geometric properties of your solution, but there are three stages which more targeted debugging might help with, they range from fastest to most in-depth and thorough.

1. In the first stage, it's worth checking whether all your points are being set up correctly, e.g. vertices are in the right locations, the watchtowers are in the right areas, etc.
2. In the second stage, the geometry may look mostly ok, but may have small errors, here you can use drawn diagrams to check that you've set all the values correctly! Using the click actions can help check each edge and each vertex is in the place you think it should be.
3. In the third stage, if all the geometry appears to be correct, set a breakpoint at the end of the split being applied and move through the next pointers, checking that everything makes sense (e.g. startVertex of the next edge is the endVertex of its preceding edge), and that all the faces are correct.

Plagiarism

This is an individual assignment. The work must be your own.

While you may discuss your program development, coding problems and experimentation with your classmates, you must not share files, as this is considered plagiarism.

If you refer to published work in the discussion of your experiments, be sure to include a citation to the publication or the web link.

“Borrowing” of someone else’s code without acknowledgment is plagiarism. Plagiarism is considered a serious offense at the University of Melbourne. You should read the University code on Academic integrity and details on plagiarism. Make sure you are not plagiarizing, intentionally or unintentionally.

You are also advised that there will be a C programming component (on paper, not on a computer) in the final examination. Students who do not program their own assignments will be at a disadvantage for this part of the examination.

Late Policy

The late penalty is 10% of the available marks for that project for each day (or part thereof) overdue. Requests for extensions on medical grounds will need to be supported by a medical certificate. Any request received less than 48 hours before the assessment date (or after the date!) will generally not be accepted except in the most extreme circumstances. In general, extensions will not be granted if the interruption covers less than 10% of the project duration. Remember that departmental servers are often heavily loaded near project deadlines, and unexpected outages can occur; these will not be considered as grounds for an extension.

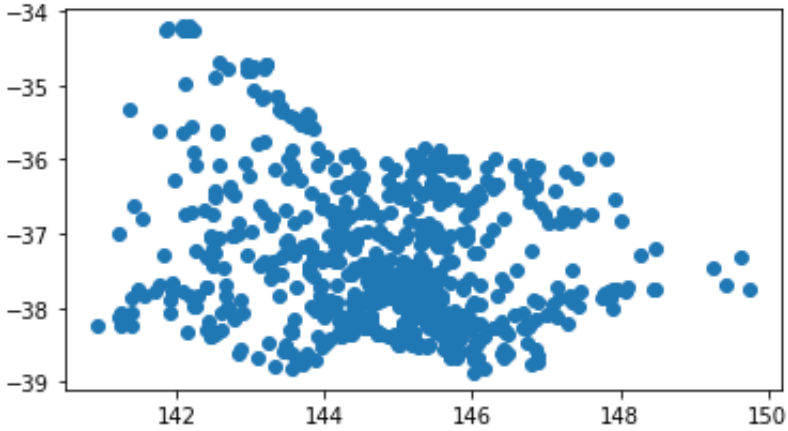
Students who experience difficulties due to personal circumstances are encouraged to make use of the appropriate University student support services, and to contact the lecturer, at the earliest opportunity.

Finally, we are here to help! There is information about getting help in this subject on the LMS. Frequently asked questions about the project will be answered on Ed.

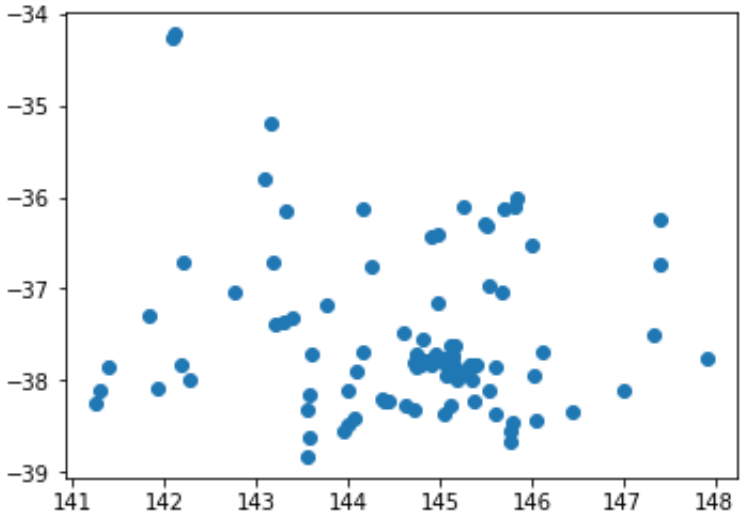
Dataset Download

Three watchtower datasets are provided:

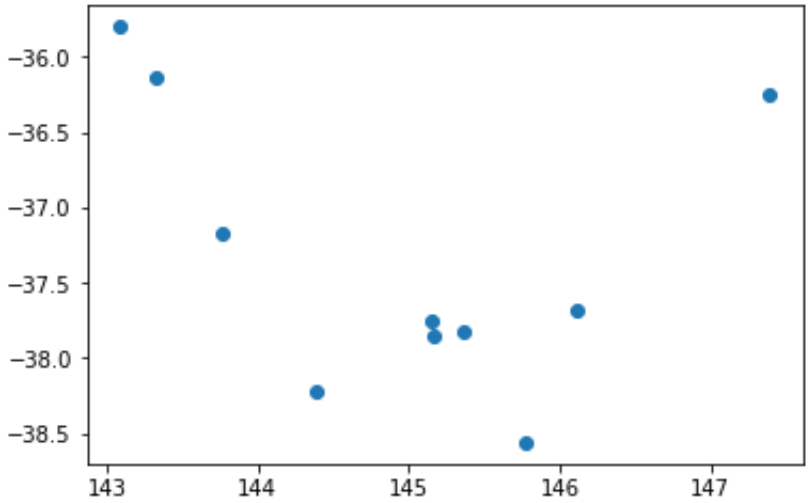
- full_dataset.csv - 1090 watchtower points



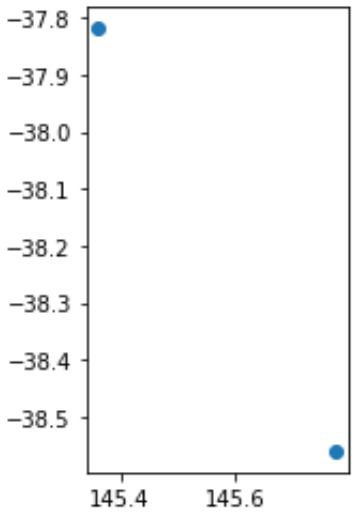
- s100ds.csv - 100 watchtower points



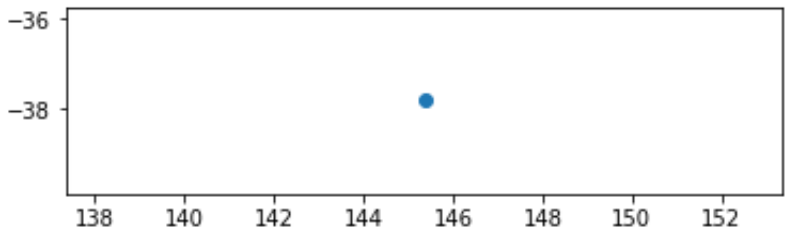
- s10ds.csv - 10 watchtower points



- s2ds.csv - 2 watchtower points



- s1ds.csv - 1 watchtower point



In addition, two base polygons (polygon_square.txt and polygon_irregular.txt) are provided, with two different sequences of splits for polygon_square.txt and five different sequences of splits for polygon_irregular.txt.

NOTE: victoria_polygon.txt covers 49152 fewer people than victoria_square.txt in the full dataset. WT3500QYTWN, WT3500GAEKD, WT3500WTTCL, WT3500TTBPR, WT3496CCVTX, WT3498JRZUR, WT3498PHNAU, WT3498MDTSF, WT3498RRAYK, WT3505LDQUW, WT3506XUEWF are the list of watch towers which fall outside the polygon.

Assignment Submission

Upload your solution here!

Testing for the first 8 marks are here. =)