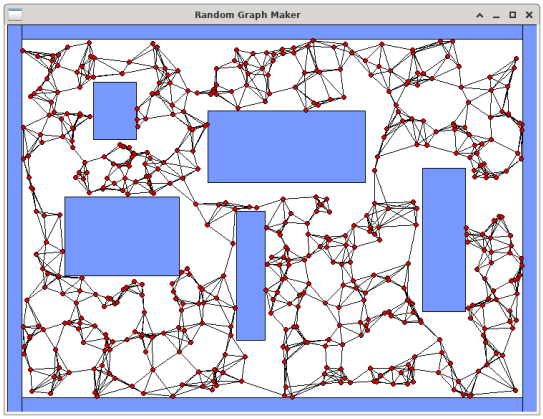
COMP2401 - Assignment #4

(Due: **Wednesday, March 13th, 2024 @ 11pm**)

In this assignment, you will gain additional practice using structs, pointers and dynamic memory allocation while creating a graph that makes use of Linked-Lists.



In the world of robotics, it is ALWAYS a good idea to program a robot to move around in its environment without hitting things. If a map of the environment is given, a robot can plan paths around the obstacles. There are many ways to perform path planning and some of the solutions involve creating a graph that can be traversed. In this assignment, we will create a random graph with vertices connected to their **k**-nearest neighbours, for some value of **k**. We will run our code on five fixed environments that are composed of rectangular obstacles. These environments will be given to you. The bottom left of the environments is always coordinate (0, 0).

To begin the assignment, you should download the following files:

- **display.c** and **display.h** – code for displaying the environment and graph.
- **obstacle.h** – structures required for this assignment.
- **graphMaker.c** – your code will be mostly written here (but there is no **main** function).
- **graphTester.c** – this is the test program that you will run.

You MUST not alter the **display.c**, **display.h** nor **obstacle.h** files. You are also NOT allowed to alter the **structs** defined in the **obstacle.h** file, NOR are you allowed to create any additional structs on this assignment. You MUST NOT CREATE ANY STATIC ARRAYS on this assignment.

***** ALERT ***** You must write your code in a modular/logical way by making helper functions/procedures for various parts of the code so that your **createGraph()** procedure (see parts 2 to 4) is **not more than 50 lines long** including comments and blank lines.

(1) You MUST first create a proper **makefile** that defines the proper dependencies, compiles the files and creates an executable called **graphTester**. You will need to include the **-lX11** library (that's "minus L X eleven") when creating the executable file. The **make clean** command should also work properly. The program must be run by supplying 3 command-line parameters as follows:

```
./graphTester <V> <K> <E>
```

Here, **<V>** is a number from 20 to 2000 (inclusively) which represents the number of vertices to add to the graph. **<K>** is a number between 1 and 25 (inclusively) that indicates the number of nearest-neighbours that each vertex will attempt to be connected to in the graph. **<E>** is a number from 1 to 5 (inclusively) representing the environment number to use. If your make file works properly, you should be able to run the program by supplying any numbers at this time, as long as they are within the proper range. The program should display the environment that you chose in a graphical window … and then wait for you to close the window.

(2) Your task will be to write all your code in the **graphMaker.c** file. Currently, there are just two blank procedures there and there is also a helper function called **linesIntersect()** that you will use. Your first task will be to write code in the **createGraph()** procedure so that it creates the vertices of the graph.

Look at the **Environment** typedef in the **obstacles.h** file. It contains a **numVertices** attribute which has been set from the command line and indicates how many vertices you need to create. The **vertices** attribute will point to a dynamically-allocated array of vertex pointers. You will need to allocate the array and set that **vertices** attribute to point to it. Then you need to create the **numVertices** vertices by allocating each vertex dynamically as well.

Look at the **Vertex** typedef in the **obstacles.h** file. Each vertex has an **x** and **y** coordinate as well as a pointer to a list of **neighbours**.

For each vertex, you must choose a random **x** value within the range of **0** to **maximumX** and a **y** value between **0** and **maximumY**. These maximums are attributes of the Environment structure. You must also ensure that the chosen (**x**, **y**) values are not inside (nor on the boundary of) any of the environment's obstacles. Notice that the environment has an **obstacles** array attribute. If you look at the **graphTester.c** file, you will notice that this array has already been created for you according to the environment selected via command-line arguments. Each **Obstacle** is assumed to be a rectangle with its top-left corner being an (**x**, **y**) point and having width **w** and height **h**.

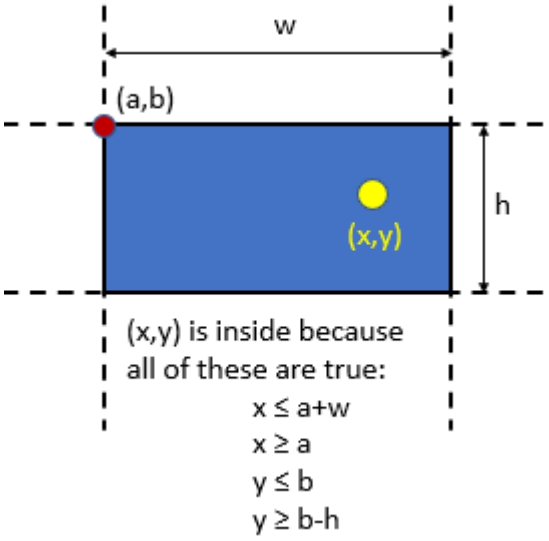
The diagram here shows how to tell whether or not a point **(x, y)** is inside the rectangle defined with **(a, b)** being the top-left corner.

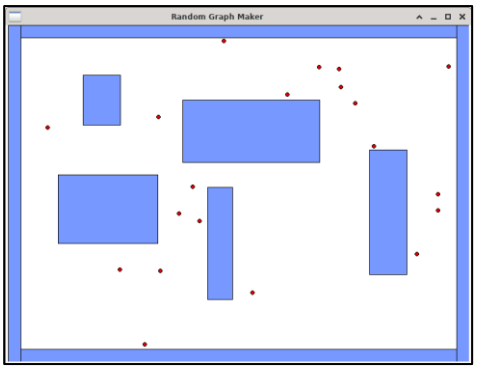
Once you have this completed, run your program as follows: `./graphTester 20 1 1` You should see something similar to what is shown here, except that your vertices will be in different locations. You should be able to count **20** red circles every time and no circle centers should be inside of a blue obstacle. It is possible, however, that a portion of the circle will be inside of an obstacle, but the very center pixel of the circle should NEVER be inside of an obstacle. Run again using `./graphTester 2000 1 1` and ensure that no circle centers are inside any obstacles. You may have to click on the window and move it a bit to ensure that the window displays all the vertices.

```
typedef struct {
  unsigned short  k;
  unsigned short  maximumX;
  unsigned short  maximumY;
  Obstacle        *obstacles;
  unsigned short  numObstacles;
  Vertex          **vertices;
  unsigned short  numVertices;
} Environment;


typedef struct vert {
  short           x;
  short           y;
  struct neigh  *neighbours;
} Vertex;


typedef struct obst {
  short       x;
  short       y;
  short       w;
  short       h;
} Obstacle;
```

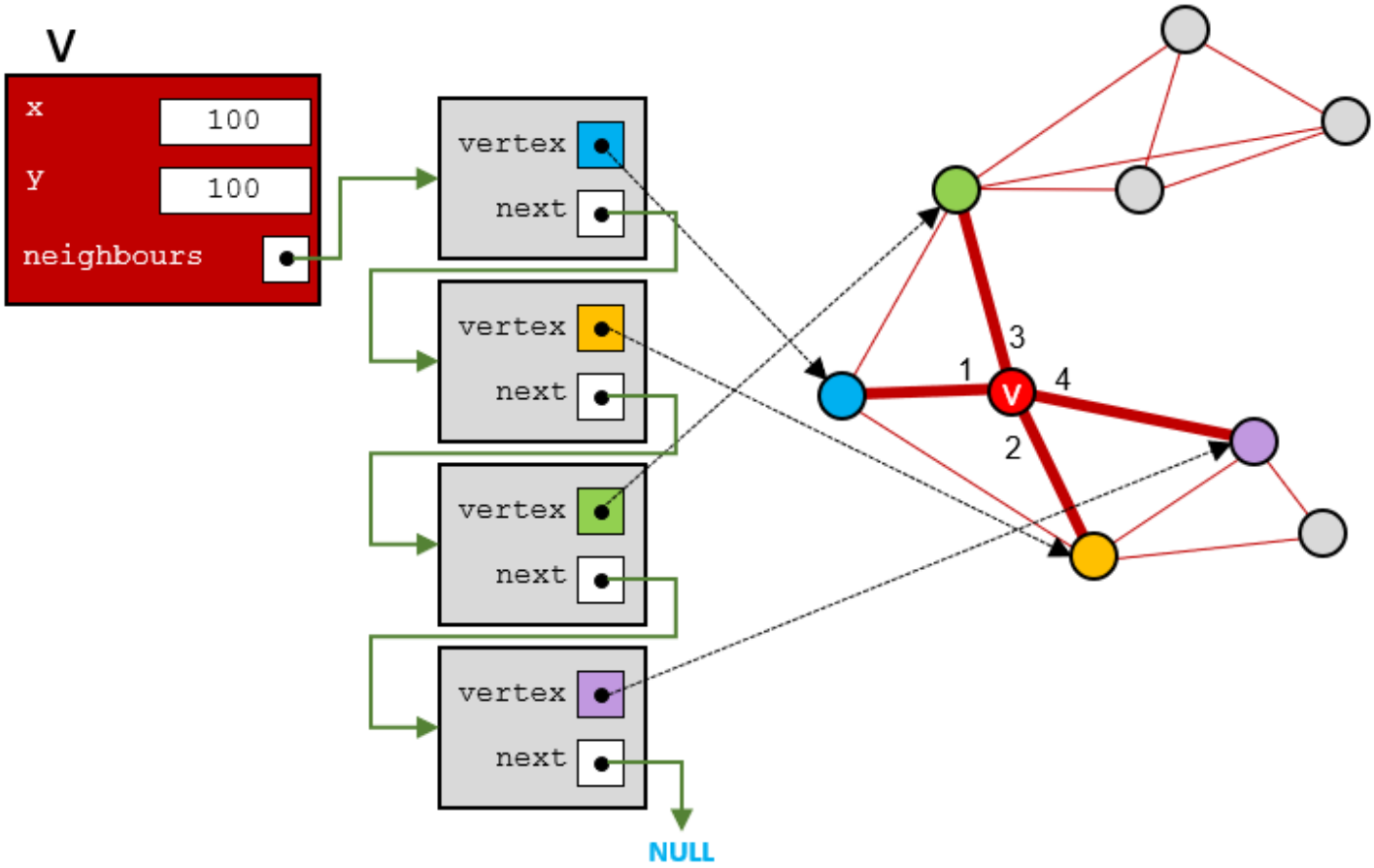




(3) Now it is time to create the graph edges. There will be no array to store the edges. Instead, each vertex will store a linked-list of the vertices that it is connected to (i.e., that vertices *neighbours*). Each vertex has an attribute called **neighbours** that should point to the head of a singly-linked-list of **Neighbour** types. The **Neighbour** type keeps a pointer to the **vertex** it represents as well as a pointer to the **next** neighbour in the list.

```
typedef struct neigh {
  Vertex        *vertex;
  struct neigh  *next;
} Neighbour;
```

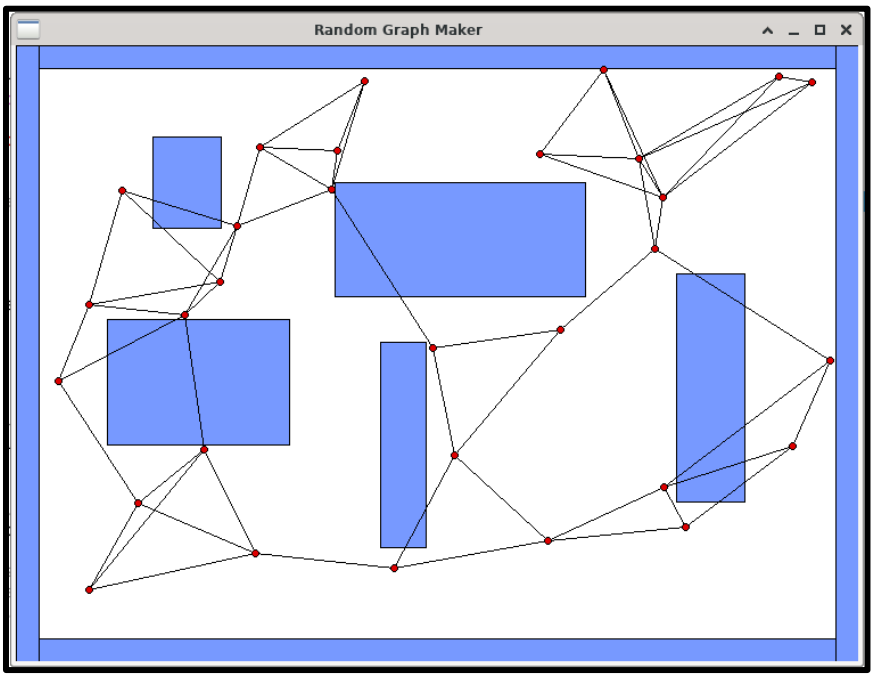
For each vertex **v**, you will go through all other vertices and find the **k**-closest vertices to **v**. The **k** value is chosen from the command-line-arguments and has been stored in the Environment struct. Consider vertex **v** in the graph below on the right. On the left, it shows the **Vertex** struct that represents it. The neighbours attribute points to the head of the linked-list that stores the **k**-nearest neighbours of **v**, where **k** = **4**. Although the order of the neighbours does not matter, it is shown such that the closest one to **v** (i.e., the blue one) is the head of the list.



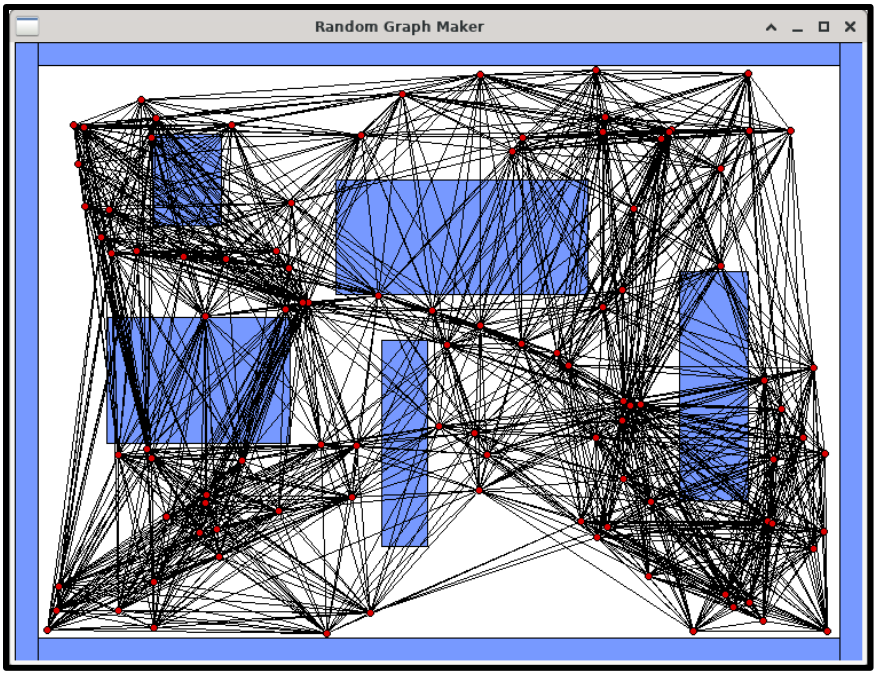
When checking for the nearest neighbours you will need to compute the distance between two vertices. You need not compute the exact distance because we don't need that value. We just need to find the closest neighbours. So, it is enough to compute the square of the distance between the points. This is faster since we do not have to compute the square root operation. The square of the distance between two points (x_1,y_1) and (x_2,y_2) is $(x_2-x_1)(x_2-x_1) + (y_2-y_1)(y_2-y_1)$. You can use whatever algorithm you want to determine the k-nearest neighbours, but simplest is best.

Once you have this completed, run your program as follows: `./graphTester 30 3 1` You should see something similar to what is shown here, except that your vertices will be in different locations. You may have to move the window once to make sure that all the vertices are

displayed. Each vertex should have **at least 3** edges connected to it. Some vertices have more than three edges connected to them. That is because in addition to that vertex's **3**-nearest neighbours, other vertices have that vertex as their neighbour, so they connect to it as well.



Run your program again as follows: `./graphTester 100 25 1` You should see something "similar to" what is shown below. Each vertex will have exactly **25** nodes in its **neighbours** list.

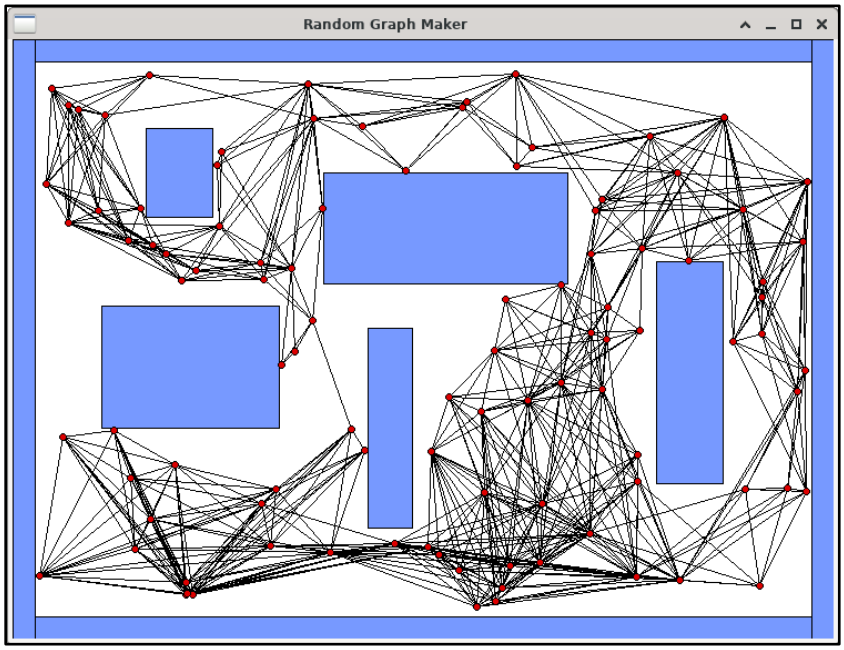


(4) Now we need to make sure that we DO NOT add any edges to the graph that cross obstacles. In your code, before you decide to add a neighbour to a vertex, you will need to first make sure that the edge that would join them does not cross any obstacles. If it does, then you will not add the neighbour. That means, it is possible that a vertex will end up with zero neighbours. If, for example, we find the **25**-nearest neighbours for a vertex **v** and **18** of those neighbours would

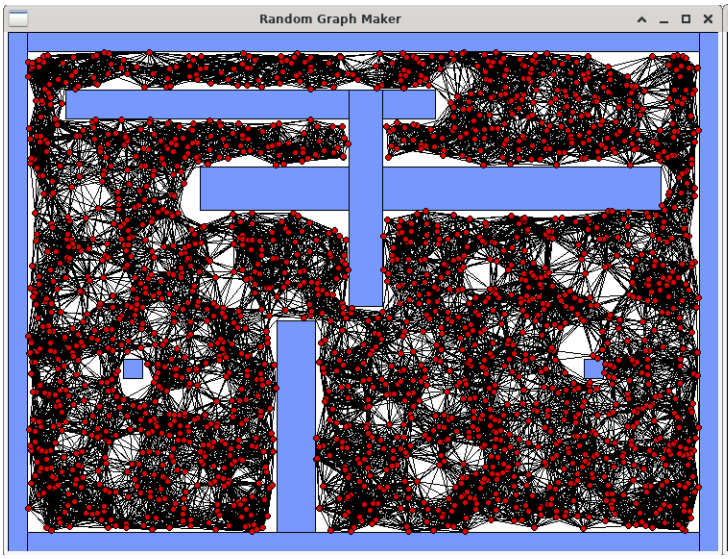
result in an edge that would cross an obstacle, then we only add the **7** neighbours to **v** that do not intersect anything, resulting in **v** having a **neighbours** list with only **7** nodes in it.

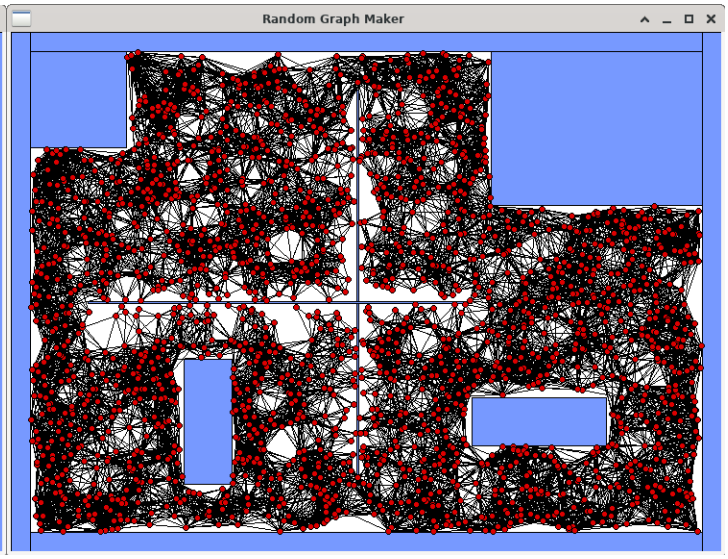
To do this, you should make use of the **linesIntersect()** function provided. Before adding neighbouring-vertex **w** to vertex **v**'s **neighbours** list, you will need to ensure that the line segment from **v** to **w** does not intersect any of sides of any rectangular obstacle in the environment's **obstacles** list.

Once you get your code written, run your program again as before: `./graphTester 100 25 1` You should see something "similar to" what is shown below. None of the edges should intersect any blue obstacle. Run it many times.

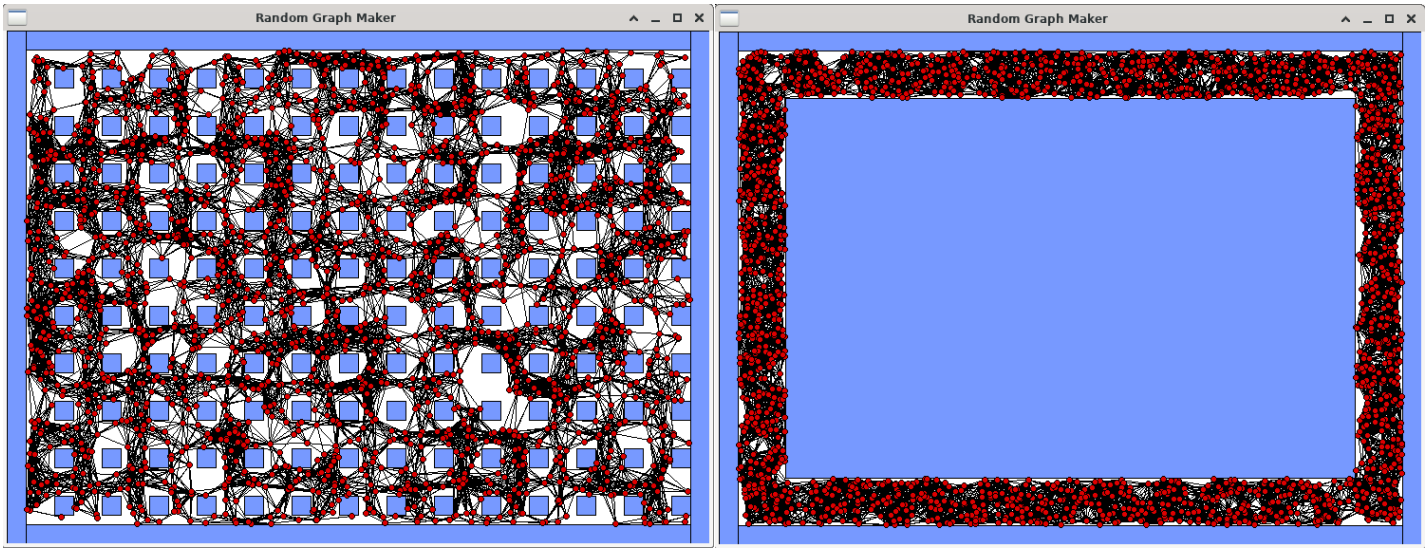


Try out a some of the other environments. Here are environments **2** & **3** with **2000** vertices and **k**=**25**:





Here are environments **4** & **5** with **2000** vertices and **k**=**25**:



(5) Finally, write code in the **cleanupEverything()** procedure so that your program has no memory leaks **nor errors**. Use `valgrind –leak-check=yes ./graphTester 2000 25 1` to do this … although you should try various combinations of grid sizes and environments as well.

IMPORTANT SUBMISSION INSTRUCTIONS:

Submit:

1. A **Readme** text file containing
   - your name and studentNumber
   - a list of source files submitted
   - any specific instructions for compiling and/or running your code

2. Your **makefile** and ALL the **.c** and **.h** files needed to compile and run.

The code **MUST** compile and run on the course VM.

- If your internet connection at home is down or does not work, we will not accept this as a reason for handing in an assignment late ... so make sure to submit the assignment WELL BEFORE it is due !

- You WILL lose marks on this assignment if any of your files are missing. So, make sure that you hand in the correct files and version of your assignment. You will also lose marks if your code is not **written neatly with proper indentation and containing a reasonable number of comments.** See course notes for examples of what is proper indentation, writing style and reasonable commenting).
