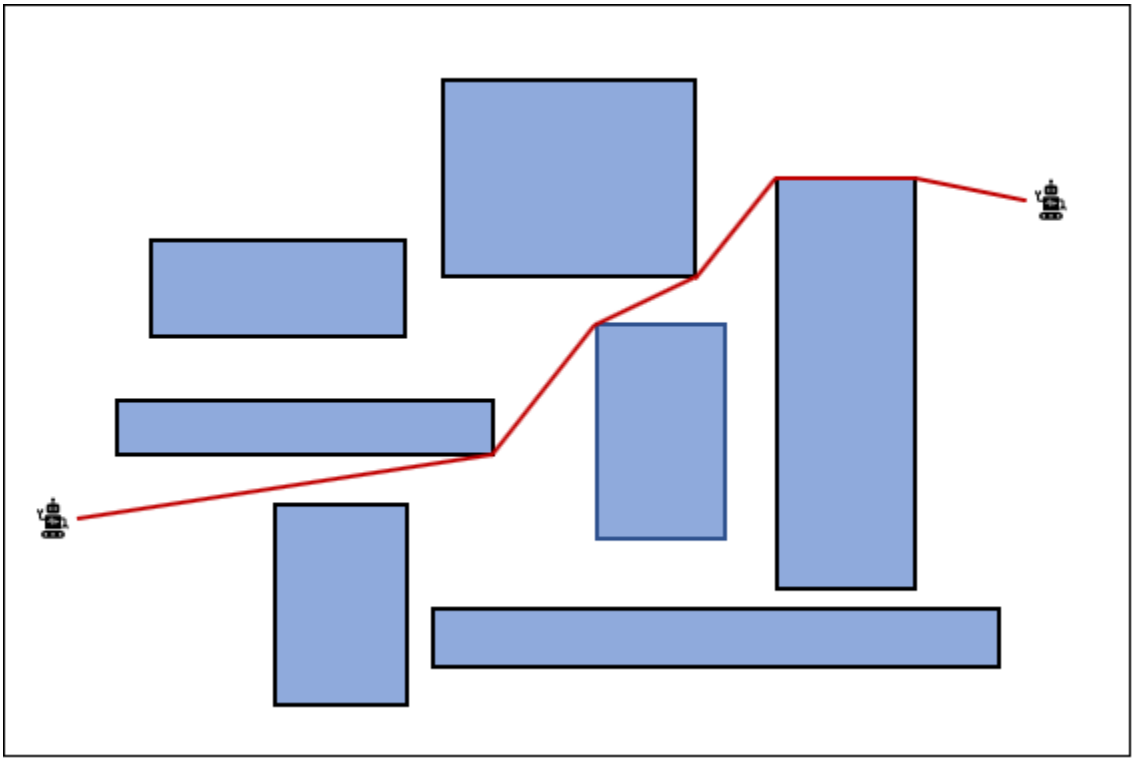
COMP2401 - Assignment #4

(Due: **Monday, November 16th, 2020 @ 6pm**)

In this assignment you will gain additional practice in using pointers to structs as well as allocating dynamic memory and creating linked lists.

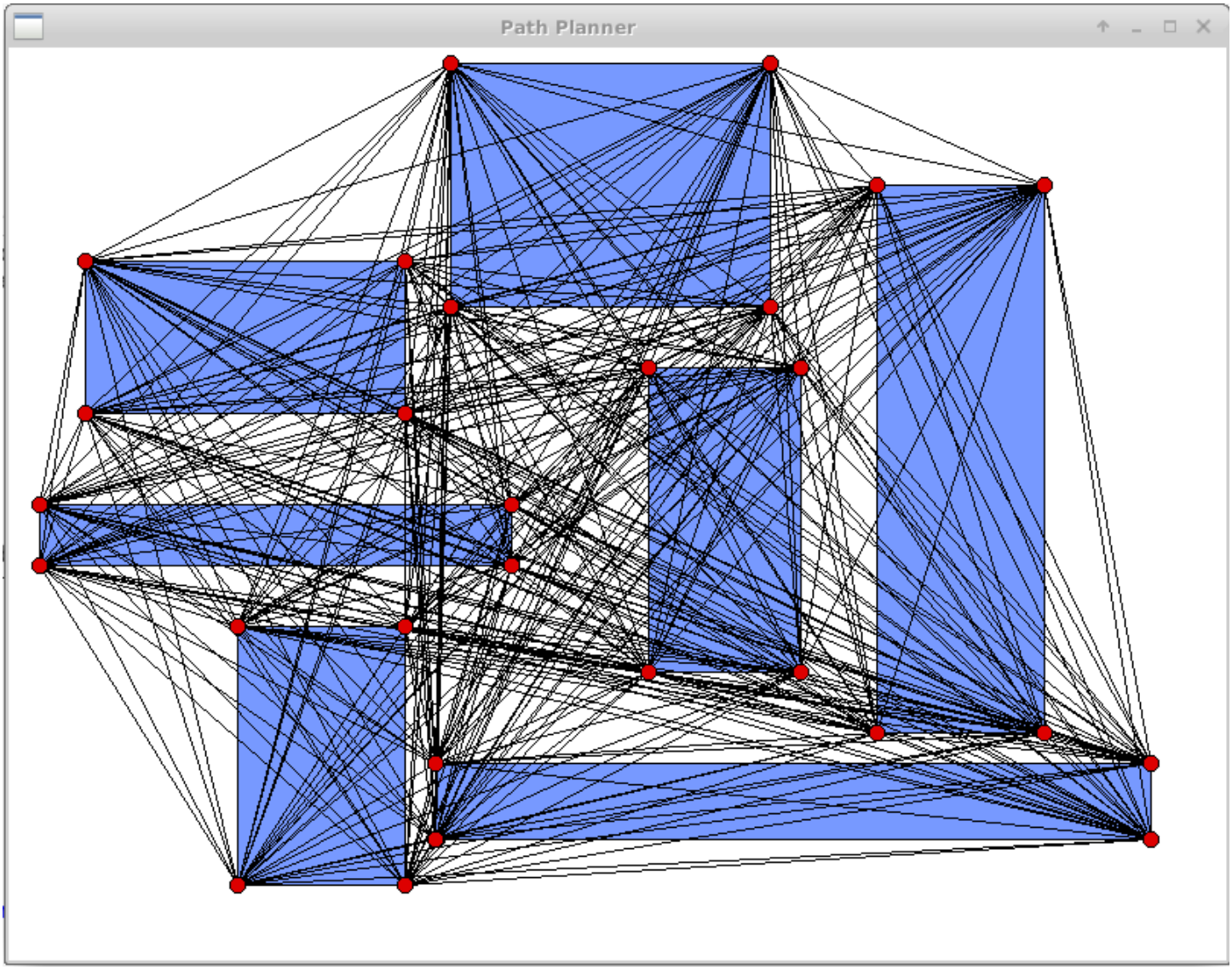
Consider a robotic environment where obstacles are represented as a set of rectangles as shown below. The robot would like to travel through the environment as efficiently as possible (i.e., shortest route) without hitting any obstacles.



To find a good route through the environment, we will assume that the robot is a single point in size (This can be easily adjusted later by "growing" the obstacles according to the robot's shape so that the robot does not bump into the obstacles when it gets close… although we will ignore this "growing" concept for this assignment). One algorithm to find the best route through the obstacles is based on computing the shortest path in a graph of edges that connects the obstacle corners.

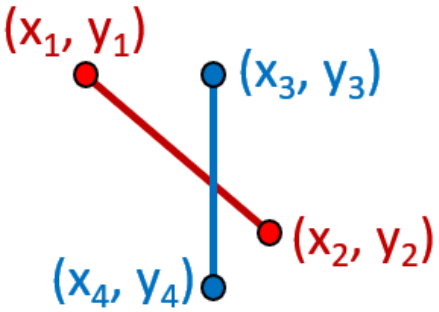
On the next page, you will see screen snapshots showing is what is known as a "complete graph" of the vertices (i.e., corners) of the obstacles. It is essentially a graph in which each vertex is connected to all other vertices. If there are **N** rectangular obstacles, then there are exactly $4\mathbf{N}*4\mathbf{N}=16\mathbf{N}^2$ edges of this complete graph. That can be a lot of edges. Finding the shortest path in this graph can be done. But it is better to reduce the graph size beforehand. As you can see, there are many edges of the graph that cross through the obstacles. Obviously, these are not edges that can be travelled on by the robot, as it will intersect/hit the obstacles.

Instead, we want to remove all edges of this graph that intersect with an obstacle. That will significantly reduce the number of edges in the graph, making it faster to find the shortest path.



How do we determine if a line segment intersects a rectangle ? We can just check to see if that edge intersects/crosses any of the obstacle edges.

Consider the two line segments shown on the right. We can compute $\mathbf{u_a}$ and $\mathbf{u_b}$ below:



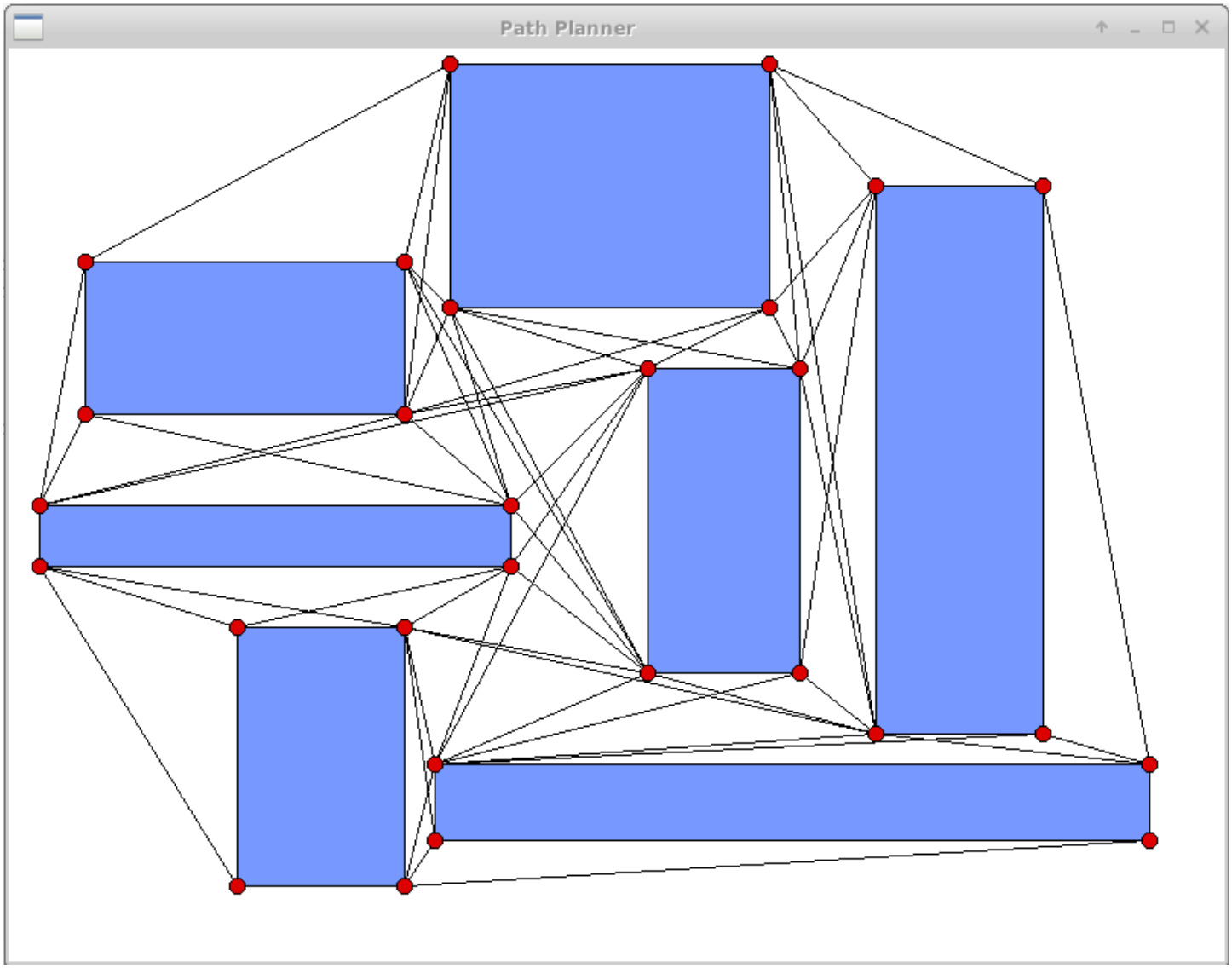
$$u_a = \frac{(x_4-x_3)(y_1-y_3) - (y_4-y_3)(x_1-x_3)}{(y_4-y_3)(x_2-x_1) - (x_4-x_3)(y_2-y_1)}$$

$$u_b = \frac{(x_2-x_1)(y_1-y_3) - (y_2-y_1)(x_1-x_3)}{(y_4-y_3)(x_2-x_1) - (x_4-x_3)(y_2-y_1)}$$

Then, the line segments intersect if $0 < \mathbf{u_a} < 1$ and $0 < \mathbf{u_b} < 1$. This does not include the case where the line segments intersect at a vertex.

We can use this simple intersection test to determine if an edge of the graph intersects a rectangle by checking all 4 sides of the rectangle. You will have to check for the special case of a edge that connects diagonal vertices of the same obstacle … as this is not a valid edge either.

Once we eliminate all edges that are invalid, we should have a graph that looks like this:



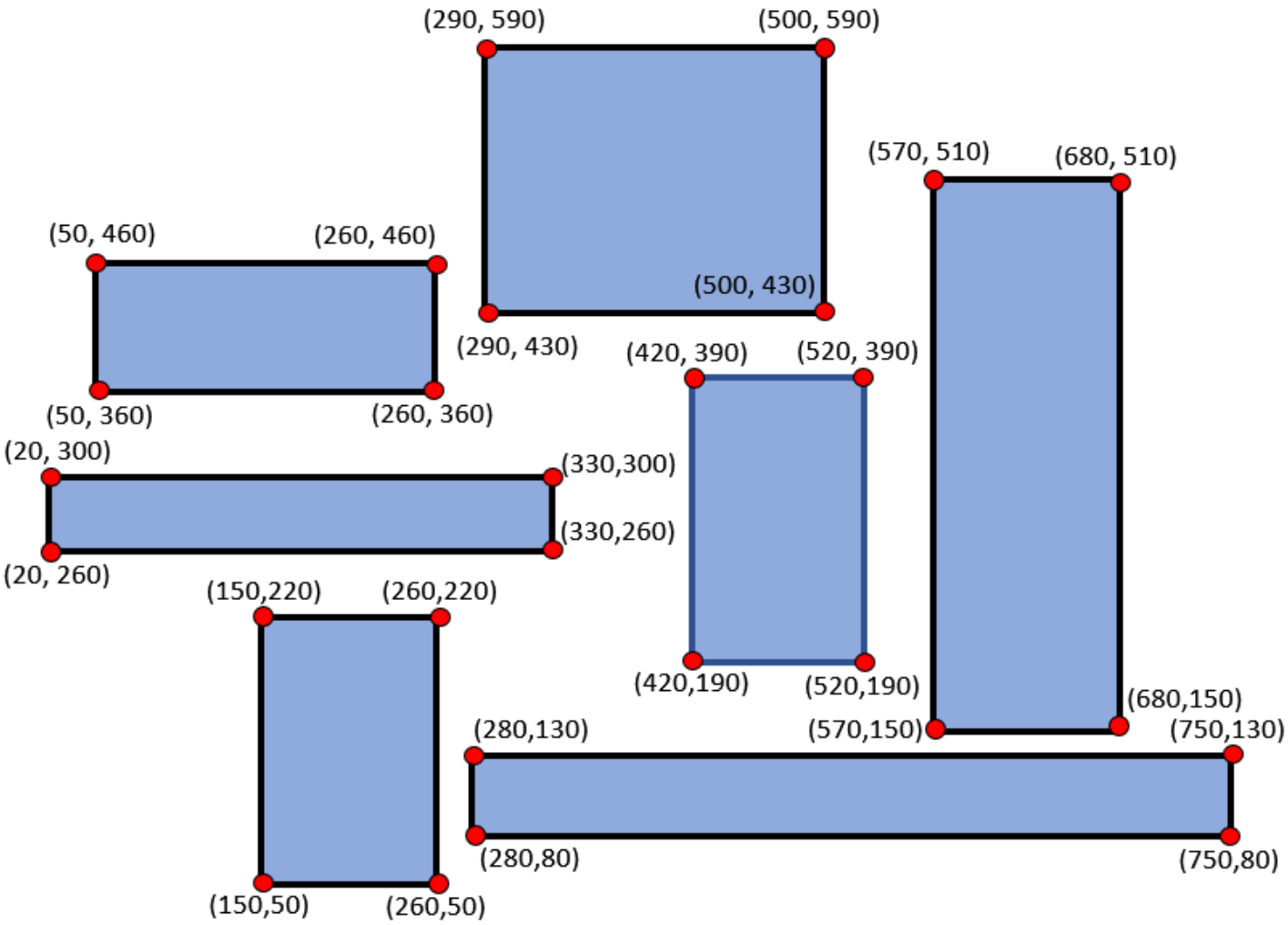
This reduced graph has 168 edges (instead of the complete one which had 756 edges). Each edge shown above is actually two edges in the graph… representing travel in both directions (i.e., one edge from vertex v1 to vertex v2 and another from vertex v2 to vertex v1). In addition, there are edges in both directions along the obstacle borders.

The shortest path from one vertex to another vertex within this graph is guaranteed to be a sequence of consecutive edges along this reduced graph. We will not be computing the shortest path in this assignment. Instead, we are just interested in computing this reduced graph, which is called a ***visibility graph***.

Follow the steps indicated to complete this assignment. You will begin with some code that already exists, which you must download.

A file called **display.c** contains code for opening a graphics window and drawing the obstacles, vertices and edges of the graph. A corresponding **display.h** file is also available for you to use. You MUST NOT modify either of these two files, yet they must be submitted along with your assignment when you hand it in.

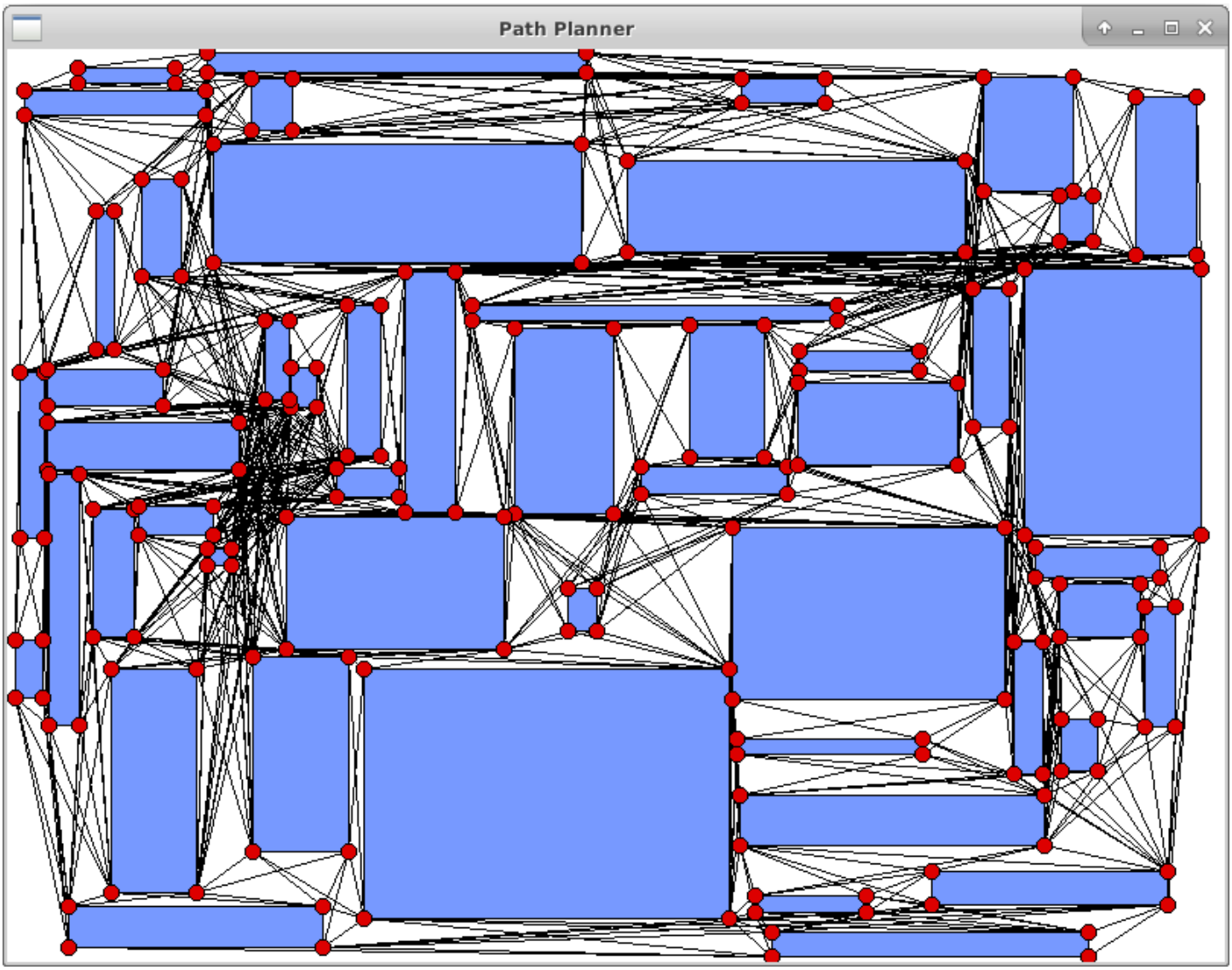
A program called **plannerTester.c** has also been written for you. It produces the example environment shown earlier. For debugging purposes, here are the obstacle coordinates:



The program creates the obstacles, and then attempts to create the vertices, the complete graph of edges, the reduced visibility graph … and then cleans up by freeing allocated memory.

Sadly, most of the code is missing. That's where you come in. The code is to be written in a file called **pathPlanner.c.** Complete the functions as needed. You MUST NOT modify this **plannerTester.c** program but you MUST include it in your final submission.

Another test program, called **bigEnvironment.c** has also been created for you. It also MUST NOT be modified and must be submitted with your assignment. It creates and environment of 50 random rectangles and then calls your pathPlanner functions as well. On the next page it shows a sample output of the produced visibility graph … although it is random each time.



- To begin, create a proper **makefile** (with a make clean as well) that compiles and generates the executables for the two test programs. You can have the **makefile** generate both test program executables by just adding an extra **gcc -o** line to the makefile. The makefile MUST make proper use of dependencies so that if a needed source code file or header file is altered, then the object file will be recompiled. You will need to include the **-lX11** library in your linking lines. That is a “minus lowercase L” at the front.

- You will need to create an **obstacles.h** file that defines the following typedefs:

    - An **Obstacle** type that maintains the x, y, width and height of a rectangular object. Examine the display.c code to see how these attributes must be defined.

    - A **Vertex** type that maintains the x,y coordinate of the vertex as well as a linked list of neighbouring vertices. That is, each vertex keeps a linked list of vertices that it connects to in the graph. You should keep pointers to the first neighbour in the list (i.e., the head) and the last neighbour in the list (i.e., the tail). The vertex should also keep a pointer to the obstacle that it belongs to (so that we can ask later if two vertices are corners on the same obstacle). Again, see the display.c code for how some of these vertex attributes must be defined. Note that the neighbours are actually **Neighbour** types … not **Vertex** types.

- A **Neighbour** type that represents an item in the linked list of vertex neighbours. It should keep a pointer to a **Vertex** that it represents, as well as a pointer to the next **Neighbour** in the list.

- Finally, an **Environment** type should be created that maintains a pointer to a dynamically-allocated array of **Obstacle** types and the number of obstacles that have been allocated. It should also maintain a pointer to a dynamically-allocated array of **Vertex** types and the number of vertices that have been allocated. You can check the test program and **display.c** code to make sure that you are defining this properly.

- You may add other things to this header file as/if you need them.

- Create all remaining necessary functions and procedures in a **pathPlanner.c** file. The functions required are indicated by the code in the test program. Note that each time the test program displays something, it waits for the user to press ENTER. You may want to write these functions in steps … by starting with the commenting out of code that you do not want to test. So, you can first make sure that all your vertices are created properly … then your complete graph … and finally your visibility graph. You should write the code as efficiently as possible. You MUST NOT hardcode any arrays. The vertices of the environment MUST be allocated dynamically, although the size will always be 4 times the number of obstacles. As for the neighbours of a vertex …they must be created individually … you may NOT create an array of neighbours at any time … they MUST be represented as a linked-list.

- Complete your code as necessary so that a call to **valgrind** returns 0 errors and 0 memory leaks. Make sure that your code is efficient and that you are making good use of your written functions.

IMPORTANT SUBMISSION INSTRUCTIONS:

Submit all of your **c source code** files as a single **tar** file containing:

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. All of your **.c source** files and all other files needed for testing/running your programs.

3. Any output files required, if there are any.

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).