Oiway.C

```
/*
Authors:             Oron Eliza     032544264
                     Mor Hadar      302676838

Project:             HW assignment 4

Description:  This file contains the main function of the program.
*/


#define _CRT_SECURE_NO_WARNINGS

#include "Server.h"
#include "Client.h"
#pragma comment(lib, "Ws2_32.lib")


int main( int argc , char* argv[])
{
       int retval = 0 ;

       if (   strcmp ( argv[1] , "server" ) == 0    )
       {
              retval = Server_Func ( argv[2] , argv[3] ,  atof ( argv[4] )  ,
argv[5]  );
              if ( retval == 1)
                     return(1);
       }
       else if ( strcmp ( argv[1] , "client" ) == 0 )
       {
              retval = Client_Func ( argv[2]  , atoi ( argv[3] ) , atoi ( argv[4]
) , atoi ( argv[5] ) , atoi ( argv[6] ) , argv[7] );
              if ( retval == 1)
                     return(1);
       }
       else
       {
              printf( "Not a valid input of client/server mode.\n");
              return(1);
       }

       return(0);

}
```

Basic types.h

```
/*
Authors:            Oron Eliza     032544264
                    Mor Hadar      302676838

Project:            HW assignment 4

Description:  This file contains the declarations of new types and constants.
*/


#ifndef BASIC_TYPES
#define BASIC_TYPES

#include <WinSock2.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#define LINE_LENGTH 1024
#define SERVER_PORT 51000
#define GPS_PORT 52000
#define LOCALHOST_STRING "127.0.0.1"
#define GPS_SEMAPHORE_NAME "GPS_SEMAPHORE"
#define GPS_PROCESS_NAME "GPS.exe"


/*
* This struct represents a single junction in the road map. The fields are as
follows:
* 1. x - The x coordinate of the junction.
* 2. y - The x coordinate of the junction.
*/
typedef struct Junction
{
       int x;
       int y;
}Junction;


/*
* This enum is used to represent the state of the current contact between the
client and the server.
* TRNS FAILED - Indicate that the connection between the client and the server has
failed.
* TRNS DISCONNECTED - Indicate that the connection between the client and the
server has lost.
* TRNS SUCCEEDED - Indicate that the connection between the client and the server
has succeeded.
* SETUP PROBLEM - Indicate that there was a problem in creating the tools to
establish the connection.
*/
typedef enum
{
       TRNS_FAILED,
```

```c
	TRNS_DISCONNECTED,
	TRNS_SUCCEEDED,
	SETUP_PROBLEM
} TransferResult_t;


/*
* Data From Server - A struct of arguments. The fields are as follows:
* 1. num of junctions - The num of junctions as written in the grpah.txt
* 2. junctions - A pointer to the array of junctions in the road map.
* 3. graph matrix - The matrix holding the traffic congestion of the rad map.
*/
typedef struct Data_From_Server
{
	int num_of_junctions;
	Junction* junctions;
	int** graph_matrix;

}Data_From_Server;


/*
* Updated Arc - A struct of arguments. The fields are as follows:
* 1. source - The first edge of the arc that has to be updated.
* 2. destination - The second edge of the arc that has to be updated.
* 3. delay - A pointer to the semaphores array.
*/
typedef struct Updated_Arc
{
	Junction source;
	Junction destination;
	int delay;
}Updated_Arc;


/*
* Single Thread Arg - A struct of arguments. The fields are as follows:
* 1. s - The accepeted socket so that the thread will be able to communicate with
single client.
* 2. client serial number - The serial number of the client.
* 3. ptr graph matrix - A pointer to the matrix holding the Traffic congestion of
the road map.
* 4. mutex graph - A pointer to mutex used to protect from race conditions on the
road map.
* 5. output file - A pointer to the server log file.
* 6. mutex file - A pointer to mutex used to protect from race conditions on the
server log file.
* 7. num of junctions - The num of junctions as written in the grpah file.
* 8. junctions - The array of junctions in the road map.
* 9. ptr quit - A pointer to flag that indicates that "quit" has being entered by
the user in the stdin.
* 10. ptr failure - A pointer to flag that indicates if a fatal error occurred
somewhere in the server program.
*/
typedef struct Single_Thread_Arg
{
	SOCKET s;
	int client_serial_number;
	int *** ptr_graph_matrix;
	HANDLE *mutex_graph;
	FILE *output_file;
	HANDLE *mutex_file;
	int num_of_junctions;
	Junction* junctions;
	BOOL *ptr_quit;
	BOOL *ptr_failure;
}Single_Thread_Arg;



#endif
```

Arguments check.h

```
/*
Authors:            Oron Eliza     032544264
                    Mor Hadar      302676838

Project:            HW assignment 4

Description:  This file contains the declarations of the functions that performs
the arguments check.
*/


#ifndef ARGUMENTS_CHECK
#define  ARGUMENTS_CHECK

#include "Bacis_Types.h"


/*
* The function checks if 3 inputs recieved from the user are valid.
*
* Input:
* -----
* 1. graph file name - The name of the input file that contains the information
about the road map.
* 2. ptr input file - A pointer to the input file (used as an additional output).
* 3. max clients - A floating point number of the max clients that the server can
serve.
* 4. server ip address - The IP address of the server.
*
* Output:
* -----
* 1. integer - Returns 1 if the arguments are not valid and 0 otherwise .
*                       **** closing file in case that the two other arguments are
invalid is made indise the function.
*/
int Arguments_Checks_Server (  char *graph_file_name , FILE** ptr_graph_file ,
double max_clients , char* server_ip_address );


/*
 * The fuction checks if the server ip addresss recieved from the user is valid.
 *
 * Input:
 * -----
 * 1. server ip address - The ip address of the server.
 *
 * Output:
 * -----
 * 1. Integer - Returns 1 if the arguments are not valid and 0 otherwise .
*/
int Arguments_Checks_Client ( char* ptr_server_ip_address );

#endif
```

Arguments_check.c

```
/*
Authors:             Oron Eliza     032544264
                     Mor Hadar      302676838

Project:             HW assignment 4

Description:  This file contains the implementation of the functions that performs
the arguments check.
*/


#include "Arguments_Check.h"


//checking the server ip address argument and in case of 'localhost' updating the
address to be "127.0.0.1"
int IP_Address_Check ( char* server_ip_address )
{
       if ( strcmp("localhost" , server_ip_address ) != 0 )
       {
              if ( inet_addr ( server_ip_address ) == INADDR_NONE)
              {
                     printf("FATAL ERROR: The string \"%s\" cannot be converted
into an ip address. Ending program.\n" , server_ip_address  );
                     return(1);
              }
       }
       else
              strcpy ( server_ip_address , LOCALHOST_STRING );
       return (0);
}

//checking the max client argument.
int Max_Clients_Check ( double max_clients )
{
       double test = 0.0 ;

       test = max_clients - (unsigned int)(max_clients);
       if ( test != 0.0 )
       {
              printf( "The max clients variable is not a positive integer!\n");
              return (1);
       }
       return(0);
}

//checking the txt file and opening it.
int Filetxt_Check (char *graph_file_name , FILE** ptr_graph_file )
{

       *ptr_graph_file = fopen ( graph_file_name , "r");
       if ( *ptr_graph_file == NULL )
       {
              printf("FATAL ERROR : Failed opening file");
              return(1);
       }
       return (0);
}


int Arguments_Checks_Server (  char *graph_file_name , FILE** ptr_graph_file ,
double max_clients , char* server_ip_address )
{
       if (    Filetxt_Check( graph_file_name , ptr_graph_file) == 1    )
              return(1);

       if (    Max_Clients_Check ( max_clients) == 1    )
       {
              fclose( *ptr_graph_file );
              return (1);
       }

       if (    IP_Address_Check ( server_ip_address) == 1    )
       {
              fclose( *ptr_graph_file );
              return(1);
       }
       return (0);
}


int Arguments_Checks_Client ( char* server_ip_address )
{
       if (    IP_Address_Check ( server_ip_address) == 1    )
              return(1);
       return(0);
}
```

Communication Tools.h

```
/*
Authors:            Oron Eliza     032544264
                    Mor Hadar      302676838

Project:            HW assignment 4

Description:  This file contains the declarations of the functions that performs
different tasks to establish and maintain the connection between the server and
the clients.
*/


#ifndef COMMUNICATION_TOOLS
#define  COMMUNICATION_TOOLS

#include "Bacis_Types.h"


/*
* The function bind the socket to a specific ip address and port.
*
* Input:
* -----
* 1. s - A pointer to the socket we wish to bind.
* 2. server ip address - The IP address of the server as a string.
* 3. port number - The port number we wish to bind the socket.
*
* Output:
* -----
* 1. integer - Returns 1 if a FATAL ERROR occured and 0 otherwise .
*/
int Bind_Func( SOCKET* s , char *server_ip_address , int port_number );


/*
* The function connect the socket to a specific ip address and port.
*
* Input:
* -----
* 1. s - A pointer to the socket we wish to connect.
* 2. server ip address - The IP address of the server as a string.
* 3. port number - The port number we wish to connect the socket.
*
* Output:
* -----
* 1. integer - Returns 1 if a FATAL ERROR occured and 0 otherwise .
*/
int Connect_Func ( SOCKET* s , char *server_ip_address , int port_number );


/*
* The function create the main server socket and force him to listen to a specific
port number and ip address.
*
* Input:
```

```
* -----
* 1. main socket - A pointer to the socket we wish to create.
* 2. server ip address - The IP address of the server as a string.
* 3. max clients - The maximum number of clients we wish the server will serve.
*
* Output:
* -----
* 1. integer - Returns 1 if a FATAL ERROR occured and 0 otherwise .
*/
int Set_Up_Server ( SOCKET* main_socket , char *server_ip_address , int
max_clinets );


/*
* The function create a client socket and connect him to a specific port number
and ip address.
*
* Input:
* -----
* 1. main socket - A pointer to the socket we wish to create..
* 2. server ip address - The IP address of the server as a string.
* 3. port number - The port number we wish to connect the socket.

* Output:
* -----
* 1. integer - Returns 1 if a FATAL ERROR occured and 0 otherwise .
*/
int Set_Up_Client ( SOCKET* main_socket , char *server_ip_address , int
port_number );


/*
* The function send a string through a specific socket and informs if the task
succeeded.
*
* Input:
* -----
* 1. str - The string we wish to send.
* 2. sd - The socket we wish to send the string through.
*
* Output:
* -----
* 1. TransferResult_t - Returns the result of the send request.
*/
TransferResult_t SendString( const char *Str, SOCKET sd );


/*
* The function receive a string through a specific socket and informs if the task
succeeded.
*
* Input:
* -----
* 1. OutputStrPtr - A pointer to the string we wish to receive.
* 2. sd - The socket we wish to receive the string through.
*
* Output:
* -----
* 1. TransferResult_t - Returns the result of the receive request.
*/
TransferResult_t ReceiveString( char** OutputStrPtr, SOCKET sd );


/*
* The function send a struct of Updated Arc through a specific socket and informs
if the task succeeded.
*
* Input:
* -----
* 1. updated arc - The struct we wish to send.
* 2. sd - The socket we wish to send the struct through.
*
* Output:
* -----
* 1. TransferResult_t - Returns the result of the send request.
*/
TransferResult_t Client_Sending_Data ( Updated_Arc updated_arc , SOCKET sd );


/*
* The function receive a struct of Data From Server through a specific socket and
informs if the task succeeded.
*
* Input:
* -----
* 1. data from server - A pointer to the struct we wish to receive.
* 2. sd - The socket we wish to receive the struct through.
*
* Output:
* -----
* 1. TransferResult_t - Returns the result of the receive request.
*/
TransferResult_t Client_Receiving_Data ( Data_From_Server *data_from_server ,
SOCKET sd );


/*
* The function send a struct of data from server through a specific socket and
informs if the task succeeded.
*
* Input:
* -----
* 1. data from server - The struct we wish to send.
* 2. sd - The socket we wish to send the struct through.
*
* Output:
* -----
* 1. TransferResult_t - Returns the result of the send request.
*/
TransferResult_t Server_Sending_Data ( Data_From_Server data_from_server , SOCKET
sd );


/*
* The function receive a struct of updated arc through a specific socket and
informs if the task succeeded.
```

```
*
* Input:
* -----
* 1. updated arc - A pointer to the struct we wish to receive.
* 2. sd - The socket we wish to receive the struct through.
*
* Output:
* -----
* 1. TransferResult_t - Returns the result of the receive request.
*/
TransferResult_t Server_Receiving_Data ( Updated_Arc *updated_arc , SOCKET sd );


#endif
```

Communication Tools.c

```c
/*
Authors:             Oron Eliza     032544264
                     Mor Hadar      302676838

Project:             HW assignment 4

Description:  This file contains the implementation of the functions that performs
different tasks to establish and maintain the connection between the server and
the clients.
*/


#include "communication_Tools.h"

int Bind_Func( SOCKET* s , char *server_ip_address , int port_number )
{
       SOCKADDR_IN service ;
       int bind_res = 0 ;

       service.sin_addr.s_addr = inet_addr ( server_ip_address );
       if ( service.sin_addr.s_addr == INADDR_NONE)
       {
              printf("FATAL ERROR: The string \"%s\" cannot be converted into an
ip address. Ending program.\n" ,server_ip_address  );
              return(1);
       }
       service.sin_family = AF_INET;
       service.sin_port = htons (port_number);

       bind_res = bind (*s , (SOCKADDR*)&service , sizeof(service) );
       if (bind_res == SOCKET_ERROR)
       {
              printf("bind() failed with error %d. Ending
program.\n",WSAGetLastError() );
              return (1);
       }
       return(0);
}

int Connect_Func ( SOCKET* s , char *server_ip_address , int port_number )
{
       SOCKADDR_IN clientService ;
       int connect_res = 0;

       clientService.sin_addr.s_addr = inet_addr ( server_ip_address);
       if ( clientService.sin_addr.s_addr == INADDR_NONE)
       {
              printf("FATAL ERROR: The string \"%s\" cannot be converted into an
ip address. Ending program.\n" ,server_ip_address  );
              return(1);
       }
       clientService.sin_family = AF_INET;
       clientService.sin_port = htons (port_number);
```

```
        connect_res = connect (*s ,  (SOCKADDR*)&clientService ,
sizeof(clientService) );
        if (connect_res == SOCKET_ERROR)
        {
                printf("connect() failed with error %d. Ending
program.\n",WSAGetLastError() );
                return (1);
        }
        return(0);
}

int Set_Up_Client ( SOCKET* ptr_main_socket , char *server_ip_address , int
port_number )
{
        WSADATA wsaData;
        int retval = 0 ;

        retval = WSAStartup (MAKEWORD(2,2) , &wsaData );
        if (retval != NO_ERROR)
        {
                printf ( "Error %ld at WSAStartup() , ending program.\n" ,
WSAGetLastError() );
                return (1);
        }

        *ptr_main_socket = socket ( AF_INET , SOCK_STREAM , IPPROTO_TCP);
        if (*ptr_main_socket == INVALID_SOCKET)
        {
                printf("Error at socket(): %ld.\n" , WSAGetLastError() );
                return(1);
        }

        retval = Connect_Func ( ptr_main_socket , server_ip_address , port_number
);
        if (retval == 1)
                return (1);
        return(0);
}

int Set_Up_Server ( SOCKET* ptr_main_socket , char *server_ip_address , int
max_clinets )
{
        WSADATA wsaData;
        int retval = 0 ;

        retval = WSAStartup (MAKEWORD(2,2) , &wsaData );
        if (retval != NO_ERROR)
        {
                printf ( "Error %ld at WSAStartup() , ending program.\n" ,
WSAGetLastError() );
                return (1);
        }

        *ptr_main_socket = socket ( AF_INET , SOCK_STREAM , IPPROTO_TCP);
        if (*ptr_main_socket == INVALID_SOCKET)
        {
                printf("Error at socket(): %ld.\n" , WSAGetLastError() );
                return(1);
        }

        retval = Bind_Func ( ptr_main_socket , server_ip_address , SERVER_PORT );
        if (retval == 1)
                return (1);

        retval = listen (  *ptr_main_socket , max_clinets  );
        if( retval == SOCKET_ERROR)
        {
                printf("Failed listening on socket, error %ld.\n" ,
WSAGetLastError() );
                //close program
                return(1);
        }
        return(0);
}

//The function send an array of char through a specific socket and informs if the
task succeeded.
TransferResult_t SendBuffer( const char* Buffer, int BytesToSend, SOCKET sd )
{
        const char* CurPlacePtr = Buffer;
        int BytesTransferred;
        int RemainingBytesToSend = BytesToSend;

        while ( RemainingBytesToSend > 0 )
        {
                BytesTransferred = send (sd, CurPlacePtr, RemainingBytesToSend, 0);
                if ( BytesTransferred == SOCKET_ERROR )
                {
                        printf("send() failed, error %d\n", WSAGetLastError() );
                        return TRNS_FAILED;
                }

                RemainingBytesToSend -= BytesTransferred;
                CurPlacePtr += BytesTransferred;
        }

        return TRNS_SUCCEEDED;
}

TransferResult_t SendString( const char *Str, SOCKET sd )
{
        int TotalStringSizeInBytes;
        TransferResult_t SendRes;

        TotalStringSizeInBytes = (int)( strlen(Str) + 1 );

        SendRes = SendBuffer(
                (const char *)( &TotalStringSizeInBytes ),
                (int)( sizeof(TotalStringSizeInBytes) ),
                sd );

        if ( SendRes != TRNS_SUCCEEDED ) return SendRes ;

        SendRes = SendBuffer(
                (const char *)( Str ),
                (int)( TotalStringSizeInBytes ),
              sd );

       return SendRes;
}

//The function receives an array of char through a specific socket and informs if
the task succeeded.
TransferResult_t ReceiveBuffer( char* OutputBuffer, int BytesToReceive, SOCKET sd
)
{
       char* CurPlacePtr = OutputBuffer;
       int BytesJustTransferred;
       int RemainingBytesToReceive = BytesToReceive;

       while ( RemainingBytesToReceive > 0 )
       {
              BytesJustTransferred = recv(sd, CurPlacePtr,
RemainingBytesToReceive, 0);
              if ( BytesJustTransferred == SOCKET_ERROR )
              {
                     printf("recv() failed, error %d\n", WSAGetLastError() );
                     return TRNS_FAILED;
              }
              else if ( BytesJustTransferred == 0 )
                     return TRNS_DISCONNECTED;

              RemainingBytesToReceive -= BytesJustTransferred;
              CurPlacePtr += BytesJustTransferred;
       }

       return TRNS_SUCCEEDED;
}

TransferResult_t ReceiveString( char** OutputStrPtr, SOCKET sd )
{
       int TotalStringSizeInBytes;
       TransferResult_t RecvRes;
       char* StrBuffer = NULL;

       if ( ( OutputStrPtr == NULL ) || ( *OutputStrPtr != NULL ) )
       {
              printf("The first input to ReceiveString() must be "
                        "a pointer to a char pointer that is initialized to NULL.
For example:\n"
                        "\tchar* Buffer = NULL;\n"
                        "\tReceiveString( &Buffer, ___ )\n" );
              return TRNS_FAILED;
       }

       RecvRes = ReceiveBuffer(
              (char *)( &TotalStringSizeInBytes ),
              (int)( sizeof(TotalStringSizeInBytes) ),
              sd );

       if ( RecvRes != TRNS_SUCCEEDED ) return RecvRes;

       StrBuffer = (char*)malloc( TotalStringSizeInBytes * sizeof(char) );
```

```
        if ( StrBuffer == NULL )
                return TRNS_FAILED;

        RecvRes = ReceiveBuffer(
                (char *)( StrBuffer ),
                (int)( TotalStringSizeInBytes),
                sd );

        if ( RecvRes == TRNS_SUCCEEDED )
                { *OutputStrPtr = StrBuffer; }
        else
        {
                free( StrBuffer );
        }

        return RecvRes;
}

TransferResult_t Client_Sending_Data ( Updated_Arc updated_arc , SOCKET sd )
{
        int total_buffer_length = 5 , *buffer_ptr = NULL;
        TransferResult_t send_res ;

        buffer_ptr = (int*)malloc( total_buffer_length * sizeof(int) );
        if (buffer_ptr == NULL )
        {
                printf("FATAL ERROR: Memory allocation failed.\n");
                return(SETUP_PROBLEM);
        }

        buffer_ptr[0] = updated_arc.source.x;
        buffer_ptr[1] = updated_arc.source.y;
        buffer_ptr[2] = updated_arc.destination.x;
        buffer_ptr[3] = updated_arc.destination.x;
        buffer_ptr[4] = updated_arc.delay;

        send_res = SendBuffer (
                        (const char*)(buffer_ptr) ,
                        (int)(total_buffer_length * sizeof(int) ) ,
                        sd);
        free(buffer_ptr);
        return(send_res);

}

TransferResult_t Client_Receiving_Data ( Data_From_Server *data_from_server ,
SOCKET sd )
{
        int total_buffer_length , i = 0 , j , z , *buffer_ptr = NULL;
        TransferResult_t rec_res ;
        rec_res = ReceiveBuffer(
                                (char*)( &total_buffer_length),
                                 (int) ( sizeof(total_buffer_length) ),
                                 sd );
        if ( rec_res != TRNS_SUCCEEDED )
                return rec_res;
        buffer_ptr = (int*) malloc(total_buffer_length * sizeof(int) ) ;
        if ( buffer_ptr == NULL )
        {
                printf("FATAL ERROR: Memory allocation failed.\n");
                return(SETUP_PROBLEM);
        }
        rec_res = ReceiveBuffer(
                                (char*)( buffer_ptr),
                                 (int) ( total_buffer_length * sizeof(int) ),
                                 sd );
        if ( rec_res != TRNS_SUCCEEDED )
        {
                free ( buffer_ptr);
                return rec_res;
        }
        data_from_server ->num_of_junctions = buffer_ptr[i];
        i++;
        data_from_server ->junctions = (Junction*) malloc(  data_from_server -
>num_of_junctions * sizeof(Junction)  );
        if ( data_from_server ->junctions == NULL )
        {
                printf("FATAL ERROR: Memory allocation failed.\n");
                free ( buffer_ptr);
                return(SETUP_PROBLEM);
        }
        for ( j = 0 ; j < data_from_server ->num_of_junctions ; j++)
        {
                data_from_server ->junctions[j].x = buffer_ptr[i];
                data_from_server ->junctions[j].y = buffer_ptr[i + 1];
                i = i + 2;
        }
        data_from_server ->graph_matrix = (int**) calloc( data_from_server -
>num_of_junctions , sizeof(int*) );
        if ( data_from_server ->graph_matrix == NULL )
        {
                printf("FATAL ERROR: Memory allocation failed\n");
                free ( buffer_ptr);
                return(SETUP_PROBLEM);
        }
        for ( j = 0 ; j < data_from_server ->num_of_junctions ; j++ )
        {
                data_from_server ->graph_matrix[j] = (int*) calloc( data_from_server
->num_of_junctions , sizeof (int) );
                if (  data_from_server ->graph_matrix[j]  == NULL  )
                {
                        printf("FATAL ERROR: Memory allocation failed\n");
                        free ( buffer_ptr);
                        return(SETUP_PROBLEM);
                }
        }
        for ( j = 0; j < data_from_server ->num_of_junctions; j++ )
        {
                for ( z = 0; z < data_from_server ->num_of_junctions; z++)
                {
                        data_from_server ->graph_matrix[j][z] = buffer_ptr[i];
                        i++;
                }
        }
        free (buffer_ptr);
        return rec_res;
```

```
}

TransferResult_t Server_Sending_Data ( Data_From_Server data_from_server , SOCKET
sd )
{
       int total_buffer_length , i = 0 , j , z , *buffer_ptr = NULL ;
       TransferResult_t send_res ;
       total_buffer_length = ( data_from_server.num_of_junctions * 2 ) + (
(data_from_server.num_of_junctions) * (data_from_server.num_of_junctions) ) + 1 ;
       send_res = SendBuffer(
                           (const char*)( &total_buffer_length),
                            (int) ( sizeof(total_buffer_length) ),
                            sd );
       if ( send_res != TRNS_SUCCEEDED )
              return send_res;
       buffer_ptr = (int*) malloc( total_buffer_length * sizeof(int) ) ;
       if ( buffer_ptr == NULL )
       {
              printf("FATAL ERROR: Memory allocation failed.\n") ;
              return(SETUP_PROBLEM) ;
       }
       buffer_ptr[i] = data_from_server.num_of_junctions ;
       i++ ;
       for ( j = 0 ; j < data_from_server.num_of_junctions ; j++)
       {
              buffer_ptr[i] = data_from_server.junctions[j].x ;
              buffer_ptr[i+1] = data_from_server.junctions[j].y ;
              i = i + 2 ;
       }
       for (  j = 0  ;   j  <  data_from_server.num_of_junctions  ;   j++  )
       {
              for (  z = 0  ;   z  <  data_from_server.num_of_junctions  ;   z++
)
              {
                     buffer_ptr[i] = data_from_server.graph_matrix[j][z] ;
                     i++ ;
              }
       }
       send_res = SendBuffer(
                           (const char*)( buffer_ptr),
                            (int) (  total_buffer_length * sizeof(int)  ),
                            sd );
       free (buffer_ptr);
       return send_res;
}

TransferResult_t Server_Receiving_Data ( Updated_Arc *updated_arc , SOCKET sd )
{
       int total_buffer_length = 5  , *buffer_ptr = NULL;
       TransferResult_t rec_res ;
       buffer_ptr = (int*) malloc( total_buffer_length * sizeof(int) );
       if (buffer_ptr == NULL )
       {
              printf("FATAL ERROR: Memory allocation failed.\n");
              return(SETUP_PROBLEM);
       }
       rec_res = ReceiveBuffer(
                           (char*)( buffer_ptr),
                            (int) ( total_buffer_length * sizeof(int) ),
                            sd );
       if ( rec_res != TRNS_SUCCEEDED )
       {
              free ( buffer_ptr);
              return rec_res;
       }
       updated_arc ->source.x = buffer_ptr[0];
       updated_arc ->source.y = buffer_ptr[1];
       updated_arc ->destination.x = buffer_ptr[2];
       updated_arc ->destination.y = buffer_ptr[3];
       updated_arc ->delay = buffer_ptr[4];
       free ( buffer_ptr);
       return rec_res;
}
```

General Tools.h

```
/*
Authors:            Oron Eliza     032544264
                    Mor Hadar      302676838

Project:            HW assignment 4

Description:  This file contains the declarations of the functions that perform
the searching junction, closing program ant printing assignments.
*/


#ifndef GENERAL_TOOLS
#define  GENERAL_TOOLS

#include "Bacis_Types.h"


/*
* Server Print Mode - An enum contains the fields as follows:
* 1. SUCCESSFULLY CONNECTED - Signal that the server successfully connected to
client.
* 2. GRAPH SENT - Signaled that the graph were sent successfully.
* 3. UPDATED ARC - Signaled that an arc were up.
* 4. CLIENT DICONNECTED - signaled that the client closed the connection.
*/
typedef enum Server_Print_Mode
{
       SUCCESSFULLY_CONNECTED,
       GRAPH_SENT,
       UPDATED_ARC,
       CLIENT_DICONNECTED

}Server_Print_Mode;


/*
* Close Status - An enum contains the fields as follows:
* 1. PLUS GPS SERVER - Signaled that we also requested to close the server socket
and the GPS socket.
* 2. PLUS SERVER - Signaled that we also requested to close the server socket.
* 3. STAND ALONE - Signaled that we requested to close only the program allocation
resources.
*/
typedef enum
{
       PLUS_GPS_SERVER = 0 ,
       PLUS_SERVER ,
       STAND_ALONE
}Close_Status;


/*
* Current Status - An enum contains the fields as follows:
* 1. SUCCESSFULLY LOGGED TO SERVER - Signaled that we successfully logged to the
server.
```

```c
* 2. FAILED TO CONNECT TO SERVER - Signaled that we faild to establish a
connection between the client and the server.
* 3. RECEIVED MAP ROAD - Signaled that we received the data from server struct
from the server.
* 4. BAD COORDINATES - Signaled that we receiveed a bad coordinates from the
operation user.
* 5. CALCULATED PATH - Signaled that we calculated the shortest path and we are
ready to print it.
* 6. SUCCESSFULLY LOOGED TO GPS - Signaled that we successfully logged to the GPS.
* 7. FAILED TO CONNECT TO GPS - Signaled that we failed to connect to the GPS.
* 8. GPS TIME - Signaled that we received the current time from the GPS.
* 9. FAILED TO RECEIVE TIME - Signaled that we failed to receive the current time
from the GPS..
* 10. FAILED TO UPDATE SERVER - Signaled that we failed to update the server(the
connection was lost).
* 11. YOU HAVE REACHED - Signaled that we have reached our destination.
* 12. FAILED TO REACH - Signaled that we failed to reach our destination.
*/
typedef enum
{
       SUCCESSFULLY_LOGGED_TO_SERVER = 0 ,
       FAILED_TO_CONNECT_TO_SERVER ,
       RECEIVED_MAP_ROAD ,
       BAD_COORDINATES ,
       CALCULATED_PATH ,
       SUCCESSFULLY_LOOGED_TO_GPS ,
       FAILED_TO_CONNECT_TO_GPS ,
       GPS_TIME ,
       FAILED_TO_RECEIVE_TIME ,
       FAILED_TO_UPDATE_SERVER ,
       YOU_HAVE_REACHED ,
       FAILED_TO_REACH
}Current_Status;


/*
* The function check if the requested source and destination junctions exsit in
the array junctions.
*
* Input:
* -----
* 1. junctions - An array that contains the junctions.
* 2. source - A junction struct that we wish to search.
* 3. destination - A junction struct that we wish to search.
* 4. source index - A pointer to the index of the source junction we will find.
* 5. destination index - A pointer to the index of the destination junction we
will find.
* 6. Num Of Junctions - The number of junctions that exist.
*
* Output:
* -----
* 1. integer - Returns 1 if a FATAL ERROR occured and 0 otherwise .
*/
int Check_If_Junction_Exist (Junction* junctions , Junction source , Junction
destination , int * source_index , int* destination_index ,
       int Num_Of_Junctions );
```

```
/*
* The function prints the current status of the server to the server LOG file and
to the sdtout.
*
* Input:
* -----
* 1. server print mode - enum defined by us indicates that status that should be
printed.
* 2. output file - A pointer to the server LOG file.
* 3. client serial number - The number of the single client.
* 4. data from client - The struct that contains the information regarding the arc
that were updated.
* 5. new weight - the new traffic conjestion of the arc.
*
* Output:
* -----
* 1. integer - Returns 1 if a FATAL ERROR occured and 0 otherwise .
*/
int Print_Server_Mode ( Server_Print_Mode server_print_mode , FILE* output_file ,
HANDLE* mutex_file, int client_serial_number ,
                                          Updated_Arc data_from_client , int
new_weight  );


/*
* The function prints the updated information about the road map before finally
closing the server program.
*
* Input:
* -----
* 1. output file - A pointer to server log file.
* 2. graph matrix - The matrix holding the Traffic congestion of the road map
* 3. num of junctions - The number of junctions in the road map.
* 4. junctions - The array of junctions in the road map.
*
* Output:
* -----
* 1. integer - Returns 1 if a FATAL ERROR occured and 0 otherwise .
*/
int Print_Graph_Into_Log_File ( FILE* output_file , int** graph_matrix , int
num_of_junctions , Junction* junctions );


/*
* The function closes and frees all the resources allocated to the program.
*
* Input:
* -----
* 1. junctions - The array of junctions in the road map.
* 2. mutex graph - A mutex used to protect from race conditions on the road map.
* 3. mutex file - A mutex used to protect from race conditions on the server log
file.
* 4. graph matrix - The matrix holding the Traffic congestion of the road map.
* 5. graph file - A pointer to the input file (used as an additional output).
* 6. output_file - A pointer to server log file.
* 7. main socket - The main listening socket of the server.
* 8. threads - The array of threads
* 9. args array - An array of the threads arguments.
* 10. max clients - The number of the max clients taht the server can serve.
* 11. num of junctions - The number of junctions in the road map.
* 12. ptr GPS semaphore - A pointer to the semaphore create by the server in order
to signal the GPS the quit recieved by the user.
*
* Output:
* -----
* 1. integer - Returns 1 if a FATAL ERROR occured and 0 otherwise .
*/
int Closing_Program_Server (Junction* junctions , HANDLE* mutex_graph , HANDLE*
mutex_file , int** graph_matrix ,
                                                 FILE *graph_file , FILE*
output_file , SOCKET main_socket , HANDLE* threads ,
                                                 Single_Thread_Arg *args_array ,
int max_clients , int num_of_junctions , HANDLE *ptr_GPS_semaphore );


/*
* The function closes and frees all the resources allocated to the program.
*
* Input:
* -----
* 1. GPS Socket - The GPS socket we wish to close.
* 2. Server Socket - The server socket we wish to close.
* 3. data from server - The struct that contains the pointers to the matrix and
array we allocated and wish to free.
* 4. close status - Signaled if we need to close and free the frogram with the
sockets or not.
* 5. shortest path arry - A pointer to the array we allocated for the junctions.
* 6. Client File - A pointer to the client file we need to close.
* 7. GPS time - A pointer to the string we allocated to receive the GPS time.
*
* Output:
* -----
* 1. integer - Returns 1 if a FATAL ERROR occured and 0 otherwise .
*/
int Close_Program_Client ( SOCKET GPS_Socket , SOCKET Server_Socket ,
Data_From_Server data_from_server , Close_Status close_status ,
                                                int *shortest_path_arry , FILE*
Client_File , char *GPS_time );


/*
* The function prints the current status of the client to the client LOG file and
to the sdtout.
*
* Input:
* -----
* 1. current status - enum defined by us indicates that status that should be
printed..
* 2. Client Log File - A pointer to the client LOG file.
* 3. junctions - The array of junctions needed to print.
* 4. shortest path array - The array of indexs that represents the shortest path
route.
* 5. Source - The junction struct of the source.
* 6. GPS new time - The time that we received from the GPS.
* 7. Route Length - The number of arcs that exist in the shortest path.
*
```

```
* Output:
* -----
* 1. integer - Returns 1 if a FATAL ERROR occured and 0 otherwise .
*/
int Print_Client_Mode ( Current_Status current_status , FILE* Client_Log_File ,
Junction *junctions , int *shortest_path_array , Junction Source ,
                                          int GPS_new_time , int Route_Length );

#endif
```

General Tools.c

```c
/*
Authors:                     Oron Eliza     032544264
                             Mor Hadar      302676838

Project:                     HW assignment 4

Description:  This file contains the implementation of the functions that performs
the searching junction, closing program and printing assignments.
*/


#include "General_Tools.h"


// constant strings that being printed in the server program.
static const char* Print_Sentence[] = {"Recieived connection from client%d\n" ,
                                        "Sent the road map to client%d\n" ,
                       "Client %d reported %d %d %d %d %d new weight is %d\n" ,
                                   "Closed connection from client%d\n" };


// constant strings that being printed in the client program.
const char* Client_Print_Status[] = {"Successfully logged into server\n" ,
                                   "Failed to connect to server\n" ,
                                   "Received the road map from server\n" ,
                                   "Bad coordinates\n" ,
                                   "Calculated path:" ,
                                   "Successfully logged into GPS\n" ,
                                   "Failed to connect to GPS\n" ,
                                   "GPS time at %d %d is %d\n" ,
                                   "Failed to receive time from GPS\n" ,
                                   "Failed to update server\n" ,
                                   "You have reached destination\n" ,
                                   "Failed to reach destination\n" };


int Check_If_Junction_Exist (Junction* junctions , Junction source , Junction
destination , int * source_index , int* destination_index ,
       int Num_Of_Junctions )
{
       int i , flag_i = 0 , flag_j = 0 ;
       for ( i = 0 ; i < Num_Of_Junctions ; i++)
       {
              if ( ( junctions[i].x == source.x ) && ( junctions[i].y == source.y
) )
              {
                     *source_index = i ;
                     flag_i = 1 ;
              }
              if ( ( junctions[i].x == destination.x ) && ( junctions[i].y ==
destination.y ) )
              {
                     *destination_index = i ;
                     flag_j = 1 ;
              }
```

```
	}

	if ( ( flag_i == 1 ) && ( flag_j == 1 ) )
		return(0);
	else
		return (1);

}


int Print_Graph_Into_Log_File ( FILE* output_file , int** graph_matrix , int
num_of_junctions , Junction* junctions )
{
	int i = 0 , j = 0 , retval = 0 ;

	char* str = NULL ;
	char temp[12];

	str = (char*)malloc ( num_of_junctions * 12 * sizeof(char) );

	// prints the number of junctions
	fprintf(output_file , "%d\n" , num_of_junctions);
	if ( retval < 0)
	{
		printf("FATAL ERROR: fprintf() failed. Ending program");
		return(1);
	}

	// prints the junctions
	sprintf( str , "%d %d" , junctions[0].x , junctions[0].y);
	for ( i = 1  ;  i < num_of_junctions  ;  i++ )
	{
		sprintf ( temp , " %d %d" ,  junctions[i].x , junctions[i].y);
		strcat ( str , temp );
	}
	strcat ( str , "\n");
	retval = fprintf( output_file , str );
	if ( retval <  0 )
	{
		printf("FATAL ERROR: fprintf() failed. Ending program");
		return(1);
	}

	// prints the road map
	for ( i = 0   ;   i < num_of_junctions   ;   i++  )
	{
		sprintf ( str , "%d" , graph_matrix[i][0] );
		for ( j = 1     ; j < num_of_junctions   ;   j++ )
		{
			sprintf ( temp , " %d" , graph_matrix[i][j]);
			strcat ( str , temp);

		}
		if ( i != num_of_junctions -1 )
		{
			strcat ( str , "\n" );
		}
		retval = fprintf( output_file , str );
```

```
                if ( retval < 0)
                {
                        printf("FATAL ERROR: fprintf() failed. Ending program");
                        return(1);
                }
        }
        free(str);
        return(0);
}


int Print_Server_Mode ( Server_Print_Mode server_print_mode , FILE* output_file ,
HANDLE* mutex_file, int client_serial_number ,
                                                Updated_Arc data_from_client , int
new_weight  )
{
        int retval = 0 ;
        DWORD res = 0;

        res = WaitForSingleObject ( *mutex_file , INFINITE );
        if (res == WAIT_FAILED )
        {
                printf("FATAL ERROR: Last error 0x%x , Ending program.\n " ,
GetLastError()  );
                return(1);
        }


        switch ( server_print_mode )
        {
        case SUCCESSFULLY_CONNECTED:
                printf(  Print_Sentence[SUCCESSFULLY_CONNECTED] ,
client_serial_number );
                retval = fprintf( output_file ,
Print_Sentence[SUCCESSFULLY_CONNECTED] , client_serial_number );
                if( retval < 0 )
                {
                        printf("FATAL ERROR: fprintf() failed. Ending program");
                        return(1);
                }
                break;

        case GRAPH_SENT:
                printf(  Print_Sentence[GRAPH_SENT] , client_serial_number);
                retval = fprintf( output_file , Print_Sentence[GRAPH_SENT] ,
client_serial_number );
                if( retval < 0 )
                {
                        printf("FATAL ERROR: fprintf() failed. Ending program");
                        return(1);
                }
                break;


        case UPDATED_ARC:
                printf(  Print_Sentence[UPDATED_ARC] , client_serial_number ,
data_from_client.source.x , data_from_client.source.y ,
data_from_client.destination.x , data_from_client.destination.y , 

data_from_client.delay , new_weight  );
                retval = fprintf( output_file ,  Print_Sentence[UPDATED_ARC] , 
client_serial_number , data_from_client.source.x , data_from_client.source.y , 

data_from_client.destination.x , data_from_client.destination.y , 

data_from_client.delay , new_weight  );
                if( retval < 0 )
                {
                        printf("FATAL ERROR: fprintf() failed. Ending program");
                        return(1);
                }
                break;

        case CLIENT_DICONNECTED:
                printf(  Print_Sentence[CLIENT_DICONNECTED] , client_serial_number);
                retval = fprintf( output_file , Print_Sentence[CLIENT_DICONNECTED] , 
client_serial_number );
                if( retval < 0 )
                {
                        printf("FATAL ERROR: fprintf() failed. Ending program");
                        return(1);
                }
                break;

        }

        res = ReleaseMutex( *mutex_file );
        if (res == 0 )
        {
                printf("FATAL ERROR: Last error 0x%x , Ending program.\n " , 
GetLastError()  );
                return(1);
        }

        return(0);
}


int Closing_Program_Server (Junction* junctions , HANDLE* mutex_graph , HANDLE* 
mutex_file , int** graph_matrix ,
                                                 FILE *graph_file , FILE* 
output_file , SOCKET main_socket , HANDLE* threads ,
                                                 Single_Thread_Arg *args_array , 
int max_clients , int num_of_junctions , HANDLE *ptr_GPS_semaphore )
{
        int i = 0 , j = 0 ;
        DWORD res = 0;
        BOOL fatal_error_Flag = FALSE ;


        // close GPS by releasing semaphore
        if ( ReleaseSemaphore( *ptr_GPS_semaphore , max_clients , NULL ) == 0)
                fatal_error_Flag = TRUE;
        if ( CloseHandle( *ptr_GPS_semaphore ) == 0 )
                fatal_error_Flag = TRUE;

        //close socket
        if (closesocket (main_socket) == SOCKET_ERROR)
                fatal_error_Flag = TRUE;


        //close junctions
        free (junctions);


        // close road map
        for ( i = 0  ;  i < num_of_junctions  ;  i++ )
                free ( graph_matrix[i] );
        free( graph_matrix );


        //close threads
        for ( i = 0  ;  i < max_clients +1  ;  i++ )
        {
                if (threads[i] != NULL )
                {
                        if (CloseHandle(threads[i]) == 0)
                            fatal_error_Flag = TRUE;
                }
        }
        free(threads);


        //close arg array
        free (args_array);


        //close mutexes
        if ( CloseHandle( *mutex_graph ) == 0 ||   CloseHandle( *mutex_file ) == 0
)
                fatal_error_Flag = TRUE;


        //close files
        if ( fclose (graph_file ) == EOF   ||   fclose( output_file ) == EOF )
                fatal_error_Flag = TRUE;


        //close communication
        if ( WSACleanup () == SOCKET_ERROR )
                fatal_error_Flag = TRUE;
        return ( fatal_error_Flag );
}


//The function receive the pointers and structs that being allocated and free
them.
void Close_Args ( Data_From_Server data_from_server , int *shortest_path_array ,
char *GPS_time )
{
        int i ;
        for ( i = 0 ;  i < data_from_server.num_of_junctions  ;  i++)
```

```
        {
                if ( data_from_server.graph_matrix[i] != NULL )
                        free( data_from_server.graph_matrix[i] );
        }
        if ( data_from_server.junctions != NULL )
                free ( data_from_server.junctions );
        if ( shortest_path_array != NULL )
                free ( shortest_path_array );
        if ( GPS_time != NULL )
                free ( GPS_time );
        return;
}


int Close_Program_Client ( SOCKET GPS_Socket , SOCKET Server_Socket ,
Data_From_Server data_from_server , Close_Status close_status ,
                                                        int *shortest_path_arry , FILE*
Client_File , char *GPS_time )
{
        int A = 0 , B = 0 , C = 0 , D  = 0;
        switch( close_status )
        {

        case PLUS_GPS_SERVER:
                if ( GPS_Socket != INVALID_SOCKET )
                        A = closesocket( GPS_Socket );
                if ( Server_Socket != INVALID_SOCKET )
                        B = closesocket( Server_Socket );
                C = WSACleanup();
                Close_Args( data_from_server , shortest_path_arry , GPS_time );
                if ( Client_File != NULL )
                        D = fclose ( Client_File );
                if  ( ( A != 0) || ( B != 0 ) || ( C != 0 ) || ( D != 0 ) )
                        return (1);
                return (0);

        case PLUS_SERVER:
                if ( Server_Socket != INVALID_SOCKET )
                        B = closesocket( Server_Socket );
                C = WSACleanup();
                Close_Args( data_from_server , shortest_path_arry , GPS_time);
                if ( Client_File != NULL )
                        D = fclose ( Client_File );
                if  ( ( B != 0 ) || ( C != 0 ) || ( D != 0 ) )
                        return (1);
                return (0);

        case STAND_ALONE:
                C = WSACleanup();
                Close_Args( data_from_server , shortest_path_arry , GPS_time);
                if ( Client_File != NULL )
                        D = fclose ( Client_File );
                if  ( ( C != 0 ) || ( D != 0 ) )
                        return (1);
                return (0);
        }
        return (0);
}
```

```
int Print_Client_Mode ( Current_Status current_status , FILE* Client_Log_File ,
Junction *junctions , int *shortest_path_array , Junction Source ,
                                          int GPS_new_time , int Route_Length )
{
        int i , ret_val ;
        switch ( current_status )
        {

        case CALCULATED_PATH:
                printf( Client_Print_Status [current_status] );
                ret_val = fprintf( Client_Log_File , Client_Print_Status
[current_status] );
                if ( ret_val < 0 )
                        {
                                printf("FATAL ERROR: fprintf() failed. Ending
program");
                                return (1);
                        }
                printf( " %d %d" , Source.x , Source.y );
                ret_val = fprintf( Client_Log_File , " %d %d" , Source.x , Source.y
);
                if ( ret_val < 0 )
                        {
                                printf("FATAL ERROR: fprintf() failed. Ending
program");
                                return (1);
                        }
                for ( i = 0 ; i < Route_Length ; i++)
                {
                        printf( " %d %d" , junctions [shortest_path_array[i] ].x ,
junctions [shortest_path_array[i] ].y );
                        ret_val = fprintf( Client_Log_File , " %d %d" , junctions
[shortest_path_array[i] ].x , junctions [shortest_path_array[i] ].y );
                        if ( ret_val < 0 )
                        {
                                printf("FATAL ERROR: fprintf() failed. Ending
program");
                                return (1);
                        }
                }
                printf("\n");
                ret_val = fprintf( Client_Log_File , "\n");
                if ( ret_val < 0 )
                        {
                                printf("FATAL ERROR: fprintf() failed. Ending
program");
                                return (1);
                        }
                return(0);

        case GPS_TIME:
                printf( Client_Print_Status [current_status] , Source.x , Source.y ,
GPS_new_time );
                ret_val = fprintf( Client_Log_File ,  Client_Print_Status
[current_status] , Source.x , Source.y , GPS_new_time );
                if ( ret_val < 0 )
```

```
                     {
                            printf("FATAL ERROR: fprintf() failed. Ending
program");
                            return (1);
                     }
              return(0);

       default:
              printf( "%s" , Client_Print_Status [current_status] );
              ret_val = fprintf( Client_Log_File , "%s" , Client_Print_Status
[current_status] );
              if ( ret_val < 0 )
                     {
                            printf("FATAL ERROR: fprintf() failed. Ending
program");
                            return (1);
                     }
              return (0);
       }
}
```

Client.h

```
/*
Authors:            Oron Eliza     032544264
                    Mor Hadar      302676838

Project:            HW assignment 4

Description:  This file contains the declaration of the function that performs
different the client assignment.
*/


#ifndef         CLIENT
#define   CLIENT

#include "Arguments_Check.h"
#include "Dijkstra's_Algorithm.h"
#include "General_Tools.h"
#include "Communication_Tools.h"



/*
* The function performs the client assignments.
*
* Input:
* -----
* 1. Arg Ip Address - The IP address of the client as a string.
* 2. Source x - The x coordinate of the source junction as a integer.
* 3. Source y  - The y coordinate of the source junction as a integer.
* 4. Destination x - The x coordinate of the Destination junction as a integer.
* 5. Destination y - The y coordinate of the Destination junction as a integer.
* 6. Client Log File_Name - The name of th client LOG file as a string.
*
* Output:
* -----
* 1. integer - Returns 1 if a FATAL ERROR or a communication problem occurred and
0 otherwise.
*/
int Client_Func ( char* Arg_Ip_Address , int Source_x , int Source_y , int
Destination_x , int Destination_y , char * Client_Log_File_Name);



#endif
```

Client.c

```
/*
Authors:             Oron Eliza     032544264
                     Mor Hadar      302676838

Project:             HW assignment 4

Description:  This file contains the implementation of the function that performs 
different tasks of the client assignment.
*/


#include "Client.h"

int Client_Func ( char* Arg_Ip_Address , int Source_x , int Source_y , int 
Destination_x , int Destination_y , char * Client_Log_File_Name)
{
       Junction Source;
       Junction Destination ;
       TransferResult_t RecvRes , send_res ;
       Updated_Arc updated_arc ;
       int  ret_val , i , num_of_arcs = 0 , delay , GPS_new_time = 0 , 
       GPS_old_time = 0 , source_index , destination_index , error_server = 0 , 
                     *shortest_path_array = NULL;
       char server_ip_address [50] , *GPS_time = NULL ;
       FILE *Client_File = NULL ;
       SOCKET Server_Socket = INVALID_SOCKET , GPS_Socket  = INVALID_SOCKET;
       Data_From_Server  data_from_server ;
       data_from_server.graph_matrix = NULL ;
       data_from_server.junctions = NULL ;
       Source.x = Source_x ;
       Source.y = Source_y ;
       Destination.x = Destination_x ;
       Destination.y = Destination_y ;



       strcpy ( server_ip_address , Arg_Ip_Address);
       ret_val = Arguments_Checks_Client(  server_ip_address  );
       if ( ret_val == 1 )
       {
              return(1);
       }

       Client_File = fopen(Client_Log_File_Name , "w" );
       if ( Client_File == NULL )
       {
              printf( "FATAL ERROR : could't create a LOG client file\n" );
              return(1);
       }

       ret_val = Set_Up_Client ( &Server_Socket , server_ip_address , SERVER_PORT 
);
       if ( ret_val == 1 )
       {
              Print_Client_Mode ( FAILED_TO_CONNECT_TO_SERVER , Client_File , 
data_from_server.junctions , shortest_path_array , Source , GPS_new_time , 0 ) ;
              ret_val = Close_Program_Client ( GPS_Socket , Server_Socket , 
data_from_server , STAND_ALONE , shortest_path_array ,  Client_File , GPS_time ) ;
              if ( ret_val == 1 )
                     printf( "FATAL ERROR : could't close Assignments\n" );

              return(1);
       }

       ret_val = Print_Client_Mode ( SUCCESSFULLY_LOGGED_TO_SERVER , Client_File , 
data_from_server.junctions , 
                                                               shortest_path_array 
,Source , GPS_new_time , 0 ) ;
       if ( ret_val == 1 )
       {
              ret_val = Close_Program_Client ( GPS_Socket , Server_Socket , 
data_from_server , PLUS_SERVER , shortest_path_array , Client_File , GPS_time ) ;
              if ( ret_val == 1 )
                     printf( "FATAL ERROR : could't close Assignments\n" );

              return(1);
       }

       RecvRes = Client_Receiving_Data ( &data_from_server , Server_Socket ) ;
       if ( RecvRes != TRNS_SUCCEEDED )
       {
              Print_Client_Mode ( FAILED_TO_CONNECT_TO_SERVER , Client_File , 
data_from_server.junctions , shortest_path_array , Source , GPS_new_time , 0 ) ;
              ret_val = Close_Program_Client ( GPS_Socket , Server_Socket , 
data_from_server , PLUS_SERVER , shortest_path_array , Client_File , GPS_time ) ;
              if ( ret_val == 1 )
                     printf( "FATAL ERROR : could't close Assignments\n" );

              return(1);
       }

       ret_val = Print_Client_Mode ( RECEIVED_MAP_ROAD , Client_File , 
data_from_server.junctions , shortest_path_array , Source , GPS_new_time , 0 ) ;
       if ( ret_val == 1 )
       {
              ret_val = Close_Program_Client ( GPS_Socket , Server_Socket , 
data_from_server , PLUS_SERVER , shortest_path_array , Client_File , GPS_time ) ;
              if ( ret_val == 1 )
                     printf( "FATAL ERROR : could't close Assignments\n" );

              return(1);
       }

       ret_val = Check_If_Junction_Exist ( data_from_server.junctions , Source , 
Destination , &source_index , &destination_index , 

       data_from_server.num_of_junctions ) ;
       if ( ret_val == 1 )
       {
              Print_Client_Mode ( BAD_COORDINATES , Client_File , 
data_from_server.junctions , shortest_path_array , Source , GPS_new_time , 0 ) ;
```

```
                ret_val = Close_Program_Client ( GPS_Socket , Server_Socket ,
data_from_server , PLUS_SERVER , shortest_path_array , Client_File , GPS_time ) ;
                if ( ret_val == 1 )
                        printf( "FATAL ERROR : could't close Assignments\n" );

                return(1);
        }

        shortest_path_array = (int*) malloc( data_from_server.num_of_junctions *
sizeof(int) );
        if ( shortest_path_array == NULL )
        {
                printf( " FATAL ERROR: memory allocation failed\n");
                ret_val = Close_Program_Client ( GPS_Socket , Server_Socket ,
data_from_server , PLUS_SERVER , shortest_path_array , Client_File , GPS_time ) ;
                if ( ret_val == 1 )
                        printf( "FATAL ERROR : could't close Assignments\n" );

                return(1);
        }

        num_of_arcs = Find_Shortest_Path ( data_from_server.graph_matrix ,
shortest_path_array , data_from_server.num_of_junctions , source_index ,

        destination_index );
         if ( num_of_arcs == -1 )
         {
                 ret_val = Close_Program_Client ( GPS_Socket , Server_Socket ,
data_from_server , PLUS_SERVER , shortest_path_array , Client_File , GPS_time ) ;
                if ( ret_val == 1 )
                        printf( "FATAL ERROR : could't close Assignments\n" );

                return(1);
         }

        ret_val = Print_Client_Mode ( CALCULATED_PATH , Client_File ,
data_from_server.junctions , shortest_path_array , Source , GPS_new_time ,
                                                              num_of_arcs ) ;
        if ( ret_val == 1 )
        {
                ret_val = Close_Program_Client ( GPS_Socket , Server_Socket ,
data_from_server , PLUS_SERVER , shortest_path_array , Client_File , GPS_time ) ;
                if ( ret_val == 1 )
                        printf( "FATAL ERROR : could't close Assignments\n" );

                return(1);
        }

        ret_val = Set_Up_Client ( &GPS_Socket , server_ip_address , GPS_PORT );
        if ( ret_val == 1 )
        {
                Print_Client_Mode ( FAILED_TO_CONNECT_TO_GPS , Client_File ,
data_from_server.junctions , shortest_path_array , Source , GPS_new_time , 0 ) ;
                ret_val = Close_Program_Client ( GPS_Socket , Server_Socket ,
data_from_server , PLUS_SERVER , shortest_path_array ,  Client_File , GPS_time ) ;
                if ( ret_val == 1 )
                        printf( "FATAL ERROR : could't close Assignments\n" );
                return(1);
        }

        ret_val = Print_Client_Mode ( SUCCESSFULLY_LOOGED_TO_GPS , Client_File ,
data_from_server.junctions ,
                                                              shortest_path_array
, Source , GPS_new_time , 0 ) ;
        if ( ret_val == 1 )
        {
                ret_val = Close_Program_Client ( GPS_Socket , Server_Socket ,
data_from_server , PLUS_GPS_SERVER ,

        shortest_path_array , Client_File , GPS_time ) ;
                if ( ret_val == 1 )
                        printf( "FATAL ERROR : could't close Assignments\n" );

                return(1);
        }

        RecvRes = ReceiveString ( &GPS_time , GPS_Socket );
        if ( RecvRes != TRNS_SUCCEEDED )
        {
                Print_Client_Mode ( FAILED_TO_RECEIVE_TIME , Client_File ,
data_from_server.junctions , shortest_path_array , Source , GPS_new_time , 0 ) ;
                Print_Client_Mode ( FAILED_TO_REACH , Client_File ,
data_from_server.junctions , shortest_path_array , Source , GPS_new_time , 0 ) ;
                ret_val = Close_Program_Client ( GPS_Socket , Server_Socket ,
data_from_server , PLUS_GPS_SERVER , shortest_path_array ,  Client_File , GPS_time
) ;
                if ( ret_val == 1 )
                        printf( "FATAL ERROR : could't close Assignments\n" );

                return(1);
        }

        GPS_new_time = atoi ( GPS_time );
        free ( GPS_time );
        GPS_time = NULL ;
        GPS_old_time = GPS_new_time;

        ret_val = Print_Client_Mode ( GPS_TIME , Client_File ,
data_from_server.junctions , shortest_path_array , Source , GPS_new_time , 0 ) ;
        if ( ret_val == 1 )
        {
                Print_Client_Mode ( FAILED_TO_REACH , Client_File ,
data_from_server.junctions , shortest_path_array , Source , GPS_new_time , 0 ) ;
                ret_val = Close_Program_Client ( GPS_Socket , Server_Socket ,
data_from_server , PLUS_GPS_SERVER ,

        shortest_path_array , Client_File , GPS_time ) ;
                if ( ret_val == 1 )
                        printf( "FATAL ERROR : could't close Assignments\n" );

                return(1);
        }

        for ( i = 0 ; i < num_of_arcs ; i++ )
        {
              RecvRes = ReceiveString ( &GPS_time , GPS_Socket );
              if ( RecvRes != TRNS_SUCCEEDED )
              {
                     Print_Client_Mode ( FAILED_TO_RECEIVE_TIME , Client_File ,
data_from_server.junctions , shortest_path_array , Source , GPS_new_time , 0 ) ;
                     Print_Client_Mode ( FAILED_TO_REACH , Client_File ,
data_from_server.junctions , shortest_path_array , Source , GPS_new_time , 0 ) ;
                     ret_val = Close_Program_Client ( GPS_Socket , Server_Socket ,
data_from_server , PLUS_SERVER , shortest_path_array ,  Client_File , GPS_time ) ;
                     if ( ret_val == 1 )
                            printf( "FATAL ERROR : could't close Assignments\n" );

                     return(1);
              }

              GPS_new_time = atoi ( GPS_time );
              free ( GPS_time );
              GPS_time = NULL ;
              delay = GPS_new_time - GPS_old_time ;
              GPS_old_time = GPS_new_time ;

              ret_val = Print_Client_Mode ( GPS_TIME , Client_File ,
data_from_server.junctions , shortest_path_array ,

       data_from_server.junctions [ shortest_path_array[i] ] , GPS_new_time , 0 )
;
              if ( ret_val == 1 )
              {
                     Print_Client_Mode ( FAILED_TO_REACH , Client_File ,
data_from_server.junctions , shortest_path_array , Source , GPS_new_time , 0 ) ;
                     ret_val = Close_Program_Client ( GPS_Socket , Server_Socket ,
data_from_server , PLUS_GPS_SERVER ,

       shortest_path_array , Client_File , GPS_time ) ;
                     if ( ret_val == 1 )
                            printf( "FATAL ERROR : could't close Assignments\n" );

                     return(1);
              }

              if ( error_server == 1 )
                     continue;

              if ( i == 0)
                     updated_arc.source = Source ;

              else
                     updated_arc.source = data_from_server.junctions [
shortest_path_array[i-1] ] ;
              updated_arc.destination = data_from_server.junctions [
shortest_path_array[i] ] ;
              updated_arc.delay = delay;

              send_res = Client_Sending_Data ( updated_arc , Server_Socket );
              if ( send_res != TRNS_SUCCEEDED )
              {
```

```
                        ret_val = Print_Client_Mode ( FAILED_TO_UPDATE_SERVER ,
Client_File , data_from_server.junctions ,

        shortest_path_array , Source , GPS_new_time , 0 ) ;
                        if ( ret_val == 1 )
                        {
                                ret_val = Close_Program_Client ( GPS_Socket ,
Server_Socket , data_from_server , PLUS_GPS_SERVER ,

        shortest_path_array , Client_File , GPS_time ) ;
                                if ( ret_val == 1 )
                                        printf( "FATAL ERROR : could't close
Assignments\n" );

                                return(1);
                        }

                        error_server = 1 ;
                }

        }

        ret_val = Print_Client_Mode ( YOU_HAVE_REACHED , Client_File ,
data_from_server.junctions ,

        shortest_path_array , Source , GPS_new_time , 0 ) ;
        if ( ret_val == 1 )
        {
                ret_val = Close_Program_Client ( GPS_Socket , Server_Socket ,
data_from_server , PLUS_GPS_SERVER ,

        shortest_path_array , Client_File , GPS_time ) ;
                if ( ret_val == 1 )
                        printf( "FATAL ERROR : could't close Assignments\n" );

                return(1);
        }

        ret_val = Close_Program_Client ( GPS_Socket , Server_Socket ,
data_from_server , PLUS_GPS_SERVER ,

        shortest_path_array , Client_File , GPS_time ) ;
        if ( ret_val == 1 )
        {
                printf( "FATAL ERROR : could't close Assignments\n" );
                return(1);
        }

        return (0);

}
```

Dijkstra’s Algorithm.h

```
/*
Authors:            Oron Eliza     032544264
                    Mor Hadar      302676838

Project:            HW assignment 4

Description:  This file contains the declaration of the function that performs the
Dijkstra Algorithm.
*/


#ifndef DIJKSTRA_ALGORITHM
#define   DIJKSTRA_ALGORITHM

#include "Bacis_Types.h"


/*
* The function find the shortest path from the source to the target junction and
insert that path to an array and return the numbers of arcs
* in that path.
*
* Input:
* -----
* 1. graph matrix - The matrix that represents the road map and the weight of the
arcs between each junction.
* 2. shortest path - An array of int that will contains the indexs of the
junctions in the shortest path (used as an additional output).
* 3. num of junctions - The number of junctions in the road map.
* 4. source - The index of the source junction in the array.
* 5. target - The index of the target junction in the array.
*
* Output:
* -----
* 1. integer - Returns the number of arcs between the junctions in the shortest
path and -1 in case of FATLA ERROR.

*/
int Find_Shortest_Path (int** graph_matrix, int* shortest_path , int
num_of_junctions, int source , int target );


#endif
```

Dijkstra’s Algorithm.c

```
/*
Authors:             Oron Eliza     032544264
                     Mor Hadar      302676838

Project:             HW assignment 4

Description:  This file contains the implementation of the functions that performs
the Dijkstra Algorithm.
*/


#include "Dijkstra's_Algorithm.h"


//Fills the shortest path array in the schematic order of the junctions in the
path
//and returns the number of arcs in the path.
int Create_Path ( int* previous , int*   shortest_path , int source , int target )
{
       int path_length = 0 , curr_vertex = target ,i = 0 ;

       while ( curr_vertex != source )
       {
              curr_vertex = previous[curr_vertex] ;
              path_length ++;
       }

       curr_vertex = target;
       shortest_path[ path_length - 1 ] = target ;
       for ( i = path_length - 2   ;   i > -1   ;   i--  )
       {
              shortest_path[i] = previous [curr_vertex ];
              curr_vertex = previous[ curr_vertex ];
       }
       return( path_length );
}


//Finds the junction with the minimum distance in the distance array which is
still in the queue
//and returns it's index in the junction array.
int Find_Min_Junction_In_Queue ( int* distance , BOOL* queue , int
num_of_junctions )
{
       int i = 0 , j = 0 , min = -1 , min_junction = 0 ;

       for ( i = 0  ;  i < num_of_junctions  ; i++ )
       {
              if (queue[i] == TRUE && distance[i] != -1 )
              {
                     min = distance[i] ;
                     min_junction = i ;
                     break;
              }
       }
```

```
        for ( j = i + 1  ;  j < num_of_junctions  ; j++  )
        {
                if (queue[j] == TRUE    &&    distance[j] != -1     &&
distance[j] < min )
                {
                        min = distance [j];
                        min_junction = j;
                }
        }

        queue[min_junction] = FALSE;
        return (min_junction);
}


int Find_Shortest_Path (int** graph_matrix , int* shortest_path , int
num_of_junctions, int source , int target )
{
        BOOL* queue = NULL ;
        int *distance = NULL , *previous = NULL , curr_vertex = 0 , alt = 0 ;
        int i=0 , path_length = 0;

        queue = (BOOL*)malloc (num_of_junctions * sizeof(BOOL)  );
        if (queue == NULL)
        {
                printf("FATAL ERROR: Memory allocation failed\n");
                return (-1);
        }

        distance = (int*)malloc (num_of_junctions * sizeof(int) );
        if (distance == NULL)
        {
                printf("FATAL ERROR: Memory allocation failed\n");
                return (-1);
        }

        previous = (int*)malloc (num_of_junctions * sizeof(int) );
        if (previous == NULL)
        {
                printf("FATAL ERROR: Memory allocation failed\n");
                return (-1);
        }

        for ( i = 0 ; i < num_of_junctions ; i++ )
        {
                if (i == source)
                        distance[i] = 0 ;
                else
                        distance[i] = -1 ;
                previous[i] = -1 ;
                queue[i] = TRUE ;
        }

        while ( ( curr_vertex = Find_Min_Junction_In_Queue(distance , queue ,
num_of_junctions) )   !=   target   )
        {
                for ( i = 0  ;  i < num_of_junctions  ;  i++  )
                {
                        if ( graph_matrix[curr_vertex][i] != -1   &&   queue[i] ==
TRUE  )
                        {
                                alt = distance[curr_vertex] +
graph_matrix[curr_vertex][i];
                                if (  alt < distance[i]   ||   distance[i] == -1   )
                                {
                                        distance[i] = alt;
                                        previous[i] = curr_vertex;
                                }
                        }
                }
        }
        path_length = Create_Path (previous , shortest_path , source , target );
        free( queue );
        free( distance );
        free( previous );
        return (  path_length   );
}
```

Server.h

```
/*
Authors:             Oron Eliza     032544264
                     Mor Hadar      302676838

Project:             HW assignment 4

Description:  This file contains the declaration of the function that performs the
server assignment.
*/


#ifndef SERVER
#define SERVER

#include "Arguments_Check.h"
#include "Converting_Text_To_Matrices.h"
#include "Server_Functions.h"


/*
* The function performs the server assignment.
*
* Input:
* -----
* 1. server ip address - The IP address of the server.
* 2. graph file name - The name of the input file that contains the information
about the road map.
* 3. max client - The number of the max clients that the server can serve.
* 4. server log file name - The name of the server log file.
*
* Output:
* -----
* 1. integer - Returns 1 if a FATAL ERROR occured and 0 otherwise .
*/
int Server_Func ( char *server_ip_address , char *graph_file_name ,  double
max_client , char *server_log_file_name);

#endif
```

Server.c

```
/*
Authors:            Oron Eliza      032544264
                    Mor Hadar       302676838

Project:            HW assignment 4

Description:  This file contains the implementation of the function that performs
the server assignment.
*/


#include "Server.h"

int Server_Func ( char *arg_ip_address , char *graph_file_name ,  double
arg_max_clients , char *server_log_file_name)
{
       Junction *junctions = NULL;
       HANDLE mutex_graph = NULL, mutex_file = NULL ;
       HANDLE GSP_semaphore = NULL ;
       int** graph_matrix = NULL ;
       char server_ip_address[50];
       int max_clients = 0 ,num_of_junctions = 0 , retval = 0 , Int = 0 ,
       client_serial_number = 1 , i = 0;
       FILE* graph_file = NULL , *output_file = NULL ;
       SOCKET main_socket = INVALID_SOCKET , accept_socket = INVALID_SOCKET;
       HANDLE *threads = NULL ;
       BOOL quit = FALSE , failure = FALSE ;
       FD_SET read_set;
       struct timeval select_timeout ;
       Single_Thread_Arg *args_array = NULL ;
       DWORD wait_res = 0 , exit_code_quit = 0 ;


       strcpy (server_ip_address , arg_ip_address);
       retval = Arguments_Checks_Server( graph_file_name , &graph_file ,
       arg_max_clients , server_ip_address  );
       if (retval == 1)
       {
              return(1);
       }

       max_clients = (int)arg_max_clients;
       output_file = fopen ( server_log_file_name , "w");
       if ( output_file == NULL )
       {
              printf("FATAL ERROR : Opening file failed. Ending program.\n");
              fclose(graph_file);
              return(1);
       }

       retval = Create_GPS_process (max_clients , server_ip_address ,
       &GSP_semaphore );
       if (retval == 1)
       {
              fclose(graph_file);
              fclose(output_file);
              return(1);
       }

       retval = Convert_Graphtext_To_Matrices( graph_file , &junctions ,
&graph_matrix ,  &num_of_junctions );
       if (retval == 1)
       {
              Closing_Program_Server (junctions , &mutex_graph , &mutex_file ,
graph_matrix ,
                                                     graph_file , output_file ,
main_socket , threads ,
                                                     args_array ,max_clients ,
num_of_junctions , &GSP_semaphore );
              return (1);
       }

       retval = Initialize_Threads_And_Mutex ( &threads , &args_array ,
max_clients , &quit , &mutex_graph , &mutex_file ,

       junctions , num_of_junctions , output_file , &graph_matrix , &failure );
       if (retval == 1)
       {
              Closing_Program_Server (junctions , &mutex_graph , &mutex_file ,
graph_matrix ,
                                                     graph_file , output_file ,
main_socket , threads ,
                                                     args_array ,max_clients ,
num_of_junctions , &GSP_semaphore );
              return(1);
       }

       retval = Set_Up_Server ( &main_socket , server_ip_address , max_clients );
       if (retval == 1)
       {
              Closing_Program_Server (junctions , &mutex_graph , &mutex_file ,
graph_matrix ,
                                                     graph_file , output_file ,
main_socket , threads ,
                                                     args_array ,max_clients ,
num_of_junctions , &GSP_semaphore );
              return(1);
       }



       select_timeout.tv_sec = 1;
       select_timeout.tv_usec = 0;

       while ( quit == FALSE  &&  failure == FALSE  )
       {
              FD_ZERO ( &read_set );
              FD_SET ( main_socket  ,  &read_set  );
              retval = select ( 0 , &read_set , NULL , NULL , &select_timeout );
              if (retval == SOCKET_ERROR)
              {
                     printf("FATAL ERROR: select() failed, error code: %d.\n" ,
GetLastError () );
```

```
            failure = TRUE;
            break;
        }
        else if( retval == 0 )
            continue;

        accept_socket = accept ( main_socket , NULL , NULL );
        if (accept_socket == INVALID_SOCKET )
        {
            printf("FATAL ERROR: Accepting connection with client failed,
error code %ld.\n" , WSAGetLastError() );
            failure = TRUE;
            break;
        }

        Int = Find_First_Unused_Theard_Slot( threads , max_clients );
        if ( Int == max_clients + 1  )
        {
            printf("No slots available for client , dropping the
connection.\n");
            retval = closesocket( accept_socket );
            if ( retval == SOCKET_ERROR )
            {
                printf("Error at closesocket(): %ld.\n" ,
WSAGetLastError() );
                failure = TRUE;
                break;
            }
            continue;
        }
        else
        {
            args_array[Int].s = accept_socket;
            args_array[Int].client_serial_number = client_serial_number;
            threads[Int] = Run_Single_Client_Thread ( Single_Client_Func ,
&(args_array[Int]) );
            if ( threads[Int] == NULL )
            {
                printf("FATAL ERROR: Last error 0x%x , Ending
program.\n " , GetLastError()  );
                failure = TRUE;
                break;
            }
            client_serial_number++ ;
        }
        failure = Check_For_Failed_Threads ( threads , max_clients );
    }

    for ( i = 1   ;   i < max_clients + 1   ;   i++ )
    {
        if (threads[i] != NULL )
        {
            wait_res = WaitForSingleObject ( threads[i] , INFINITE );
            if (wait_res == WAIT_FAILED )
            {
                failure = TRUE;
                printf("FATAL ERROR: Last error 0x%x , Ending
program.\n " , GetLastError()  );
```

```
                     }
              }
       }


       if ( failure == TRUE )
       {
              TerminateThread ( threads[0] , exit_code_quit );
              Closing_Program_Server (junctions , &mutex_graph , &mutex_file ,
graph_matrix ,
                                                        graph_file , output_file ,
main_socket , threads ,
                                                        args_array ,max_clients ,
num_of_junctions , &GSP_semaphore );
              return(1);
       }

       retval = Print_Graph_Into_Log_File ( output_file , graph_matrix ,
num_of_junctions , junctions );
       if( retval == 1 )
       {
              Closing_Program_Server (junctions , &mutex_graph , &mutex_file ,
graph_matrix ,
                                                        graph_file , output_file ,
main_socket , threads ,
                                                        args_array ,max_clients ,
num_of_junctions , &GSP_semaphore );

              return (1);
       }
       retval = Closing_Program_Server (junctions , &mutex_graph , &mutex_file ,
graph_matrix ,
                                                         graph_file , output_file ,
main_socket , threads ,
                                                          args_array ,max_clients ,
num_of_junctions , &GSP_semaphore );
       if(retval == 1)
              return(1);
       return(0);
}
```

Converting Text To Matrices.h

```
/*
Authors:            Oron Eliza     032544264
                    Mor Hadar      302676838

Project:            HW assignment 4

Description:  This file contains the declarations of the functions that performs
the converting the textual information from the input file to the different
arrays.
*/


#ifndef CONVERTING_TEXT_TO_MATRICES
#define  CONVERTING_TEXT_TO_MATRICES

#include "Bacis_Types.h"


/*
* The function convetrs the textual information from the input file to be
represented by different arrays.
*
* Input:
* -----
* 1. graph file - A pointer to input file that contains the information about the
road map.
* 2. ptr junctions - A pointer to the array of junctions in the road map (used as
an additional output).
* 3. ptr graph matrix - A pointer to the matrix holding the Traffic congestion of
the road map (used as an additional output).
* 4. ptr num of junctions - A pointer to the number of junctions in the road map.
*
* Output:
* -----
* 1. integer - Returns 1 if a FATAL ERROR occured and 0 otherwise .
*/
int Convert_Graphtext_To_Matrices(FILE* graph_file , Junction** ptr_junctions ,
int*** ptr_graph_matrix , int* ptr_num_of_junctions);


#endif
```

Converting Text To Matrices.c

```
/*
Authors:             Oron Eliza     032544264
                     Mor Hadar      302676838

Project:             HW assignment 4

Description:  This file contains the implementation of the functions that performs
the converting the textual information from the input file to the different
arrays.
*/


#include "Converting_Text_To_Matrices.h"


// The function allocate memory for the junction array and filling it with the
information from the road map written in the input file.
int Create_Junctions_Array(int num_of_junctions , Junction** ptr_junctions , FILE*
graph_file)
{
       char line[LINE_LENGTH];
       char* retstr = NULL ;
       int i = 0;

       *ptr_junctions = (Junction*) malloc(  num_of_junctions * sizeof(Junction)
);
       if (*ptr_junctions == NULL )
       {
              printf("FATAL ERROR: Memory allocation failed\n");
              return(1);
       }

       retstr = fgets(line , LINE_LENGTH , graph_file);
       if (retstr == NULL )
       {
              printf("FATAL ERROR: fgets() failed. Ending program");
              return(1);
       }
       line[  strlen(line) -1  ] = '\0' ;

       (*ptr_junctions)[0].x = atoi (  strtok(line , " ")  );
       (*ptr_junctions)[0].y = atoi (  strtok(NULL , " ")  );
       for ( i = 1  ;  i < num_of_junctions ; i++ )
       {
              (*ptr_junctions)[i].x = atoi (  strtok(NULL , " ")  );
              (*ptr_junctions)[i].y = atoi (  strtok(NULL , " ")  );
       }
       return(0);
}


// The function allocate memory for the graph matrix and filling it with the
information from the road map written in the input file.
int Create_Graph_Matrix (int num_of_junctions , int*** ptr_graph_matrix , FILE*
graph_file )
```

```
{
       char line[LINE_LENGTH];
       char *retstr = NULL ;
       int i = 0 , j = 0;

       *ptr_graph_matrix = (int**)malloc( num_of_junctions * sizeof(int*) );
       if (*ptr_graph_matrix == NULL )
       {
              printf("FATAL ERROR: Memory allocation failed\n");
              return (1);
       }
       for ( i = 0 ; i < num_of_junctions ; i++ )
       {
              (*ptr_graph_matrix)[i] = (int*)malloc ( num_of_junctions * sizeof
(int) );
              if (  (*ptr_graph_matrix)[i]  == NULL  )
              {
                     printf("FATAL ERROR: Memory allocation failed\n");
                     return (1);
              }
       }

       for ( i = 0 ; i < num_of_junctions ; i++)
       {
              retstr = fgets (line , LINE_LENGTH , graph_file);
              if (retstr == NULL )
              {
                     printf("FATAL ERROR: fgets() failed. Ending program");
                     return(1);
              }
              if ( feof(graph_file) == 0 )
              {
                     line [ strlen(line) -1 ] = '\0';
              }
              (*ptr_graph_matrix)[i][0] = atoi (  strtok( line , " ")  );
              for (  j = 1  ;  j < num_of_junctions  ;  j++  )
                     (*ptr_graph_matrix)[i][j] = atoi (  strtok( NULL , " ")  );
       }
       return (0);
}


int Convert_Graphtext_To_Matrices(FILE* graph_file , Junction** ptr_junctions ,
int*** ptr_graph_matrix , int* ptr_num_of_junctions)
{
       char line[LINE_LENGTH];
       char *retstr = NULL ;
       int line_length = 0;
       int retval=0;

       retstr = fgets (line , LINE_LENGTH , graph_file);
       if (retstr == NULL )
       {
              printf("FATAL ERROR: fgets() failed. Ending program");
              return(1);
       }
       line[  strlen(line) -1  ] = '\0' ;
       *ptr_num_of_junctions = atoi(line);
       retval = Create_Junctions_Array( *ptr_num_of_junctions  ,  ptr_junctions  ,
graph_file );
       if (retval == 1)
              return(1);

       retval = Create_Graph_Matrix ( *ptr_num_of_junctions  ,  ptr_graph_matrix ,
graph_file );
       if (retval == 1)
              return(1);

       return(0);
}
```

Server_Functions.h

```c
/*
Authors:            Oron Eliza     032544264
                    Mor Hadar      302676838

Project:            HW assignment 4

Description:  This file contains the declarations of the functions that performs
different tasks of the server assignment.
*/


#ifndef SERVER_THREAD_FUNC
#define SERVER_THREAD_FUNC

#include "Bacis_Types.h"
#include "communication_Tools.h"
#include "General_Tools.h"


/*
* The function creates the GPS semaphore anD creates the GPS process.
*
* Input:
* -----
* 1. max clients - The number of the max clients that the server can serve.
* 2. server ip address - The IP address of the server.
* 3. ptr GPS semaphore - A pointer to the semaphore create by the server in order
to signal the GPS that quit recieved by the user.
*
* Output:
* -----
* 1. integer - Returns 1 if a FATAL ERROR occured and 0 otherwise .
*/
int Create_GPS_process (int max_clients , char* server_ip_address , HANDLE
*ptr_GPS_semaphore ) ;


/*
* The function intialize: 1. The threads of the server (allocates memory for the
array, runs the quit thread and intialize all the client threads arguments.
*2. The two mutexes that protecting the graph matrix and the output file.
*
* Input:
* -----
* 1. ptr threads - A pointer to the array of threads (used as an additional
output).
* 2. ptr args - A pointer to array of Single Thread Arg (used as an additional
output).
* 3. max clients - The number of the max clients taht the server can serve.
* 4. ptr quit - A pointer to flag that indicates that "quit" has being entered by
the user in the stdin.
* 5. ptr mutex graph - A pointer to mutex used to protect from race conditions on
the road map.
* 6. ptr mutex file -  A pointer to mutex used to protect from race conditions on
the server log file.
```

```
* 7. junctions - The array of junctions in the road map.
* 8. num of junctions - The num of junctions as written in the grpah file
* 9. output file - A pointer to the server log file.
* 10. ptr graph matrix - A pointer to the matrix holding the Traffic congestion of
the road map.
* 11. ptr failure - A pointer to flag that indicates if a fatal error occurred
somewhere in the server program.
*
* Output:
* -----
* 1. integer - Returns 1 if a FATAL ERROR occured and 0 otherwise .
*/
int Initialize_Threads_And_Mutex ( HANDLE** ptr_threads , Single_Thread_Arg
**ptr_args , int max_clients ,  BOOL* ptr_quit ,
                                                           HANDLE *ptr_mutex_graph
, HANDLE *ptr_mutex_file , Junction *junctions , int num_of_junctions ,
                                                           FILE* output_file ,
int*** ptr_graph_matrix , BOOL* ptr_failure );


/*
* The function finds the first available thread in order to serve client that were
accepted.
*
* Input:
* -----
* 1. threads - The array of threads.
* 2. max clients - The number of the max clients that the server can serve.

* Output:
* -----
* 1. integer - The index in the array of the first available thread.
                        (If the return value is max_clients+1 it indicate that all
theard are occupied).
*/
int Find_First_Unused_Theard_Slot( HANDLE *threads , int max_clients );


/*
* The function runs a single thread that will communicate with a single client.
*
* Input:
* -----
* 1. (*func)(Single_Thread_Arg*) - The function that the thread will perform. The
func detailed below.
* 2. arg - A pointer to the struct of argument that the thread need. Detailed in
Basic Types module.
*
* Output:
* -----
* 1. HANDLE to the thread. The returend value is NULL if the function fails.
*/
HANDLE Run_Single_Client_Thread ( int (*func)(Single_Thread_Arg*) ,
Single_Thread_Arg  *arg);


/*
* The function checks if one or more threads finished running due FATAL ERROR.
```

```
*
* Input:
* -----
* 1. threads - The array of threads
* 2. max clients - The number of the max clients taht the server can serve.
*
* Output:
* -----
* 1. BOOL - TRUE if a FATAL ERROR occured in one of the thread and FALSE
otherwise.
*/
BOOL Check_For_Failed_Threads ( HANDLE* threads , int max_clients );


/*
* The function perform the single client communication assignment.
*
* Input:
* -----
* 1. arg - A pointer to the struct of argument that the thread need. Detailed in
Basic Types module.
*
* Output:
* -----
* 1. integer - Returns 1 if a FATAL ERROR occured and 0 otherwise .
*/
int Single_Client_Func ( Single_Thread_Arg   *arg );


#endif
```

Server_Functions.c

```
/*
Authors:            Oron Eliza     032544264
                    Mor Hadar      302676838

Project:            HW assignment 4

Description:  This file contains the implementation of the functions that performs
different tasks of the server assignment.
*/


#include "Server_Functions.h"


int Create_GPS_process (int max_clients , char* server_ip_address , HANDLE
*ptr_GPS_semaphore )
{
       PROCESS_INFORMATION proc_info;
       SECURITY_ATTRIBUTES security;
       char command_line[50];
       BOOL retval = 0;
       STARTUPINFO   startinfo = { sizeof( STARTUPINFO ), NULL, 0 };

       security.nLength = sizeof(security);
       security.lpSecurityDescriptor = NULL ;
       security.bInheritHandle = TRUE ;
       *ptr_GPS_semaphore = CreateSemaphore(&security , 0 , max_clients ,
GPS_SEMAPHORE_NAME );
       if ( *ptr_GPS_semaphore == NULL )
       {
              printf("FATAL ERROR: CreatSemaphore() failed, error code: %d.\n" ,
GetLastError () );
              return(1);
       }

       sprintf (command_line , "%s %s %d" , GPS_PROCESS_NAME , server_ip_address ,
max_clients);
       retval = CreateProcess( NULL, command_line, NULL, NULL, FALSE,
CREATE_NEW_CONSOLE,
                                                NULL, NULL, &startinfo,
       &proc_info );
       if (retval == 0 )
       {
              printf("FATAL ERROR: CreateProcess() failed, error code: %d.\n" ,
GetLastError () );
              CloseHandle(*ptr_GPS_semaphore);
              return(1);
       }
       return (0);
}


//Performs the reading "quit" from stdin task. Is being called by the
Initialize_Threads_And_Mutex function.
void Quit_Thread ( BOOL* ptr_quit )
```

```
{
       char line [6];
       while ( *ptr_quit  == FALSE )
       {
              gets (line);
              if ( strcmp(line , "quit") == 0 )
                     *ptr_quit = TRUE;
       }
       return ;
}


// Creates an handle for a single thread running the Quit Func.
HANDLE Run_Quit_Single_Thread ( void (*func)(BOOL*) , BOOL *ptr_quit)
{
       return CreateThread ( NULL  ,  0  ,  (LPTHREAD_START_ROUTINE)func  ,
ptr_quit  ,  0  ,  NULL  );
}


HANDLE Run_Single_Client_Thread ( int (*func)(Single_Thread_Arg*) ,
Single_Thread_Arg  *arg)
{
       return CreateThread ( NULL  ,  0  ,  (LPTHREAD_START_ROUTINE)func  ,  arg
,  0  ,  NULL  );
}


int Initialize_Threads_And_Mutex ( HANDLE** ptr_threads , Single_Thread_Arg
**ptr_args , int max_clients ,  BOOL* ptr_quit ,
                                                          HANDLE *ptr_mutex_graph
, HANDLE *ptr_mutex_file ,Junction *junctions , int num_of_junctions ,
                                                          FILE* output_file ,
int*** ptr_graph_matrix , BOOL* ptr_failure )
{
       int i = 0;
       *ptr_threads = (HANDLE*)malloc ( ( max_clients + 1 ) * sizeof(HANDLE) );
       *ptr_args = (Single_Thread_Arg*)malloc( ( max_clients + 1 ) *
sizeof(Single_Thread_Arg) );
       if ( *ptr_threads == NULL || *ptr_args == NULL  )
       {
              printf("FATAL ERROR : Memory allocation failed.\n");
              return(1);
       }

       (*ptr_threads) [0] = Run_Quit_Single_Thread ( Quit_Thread, ptr_quit ) ;
       if ( (*ptr_threads) [0] == NULL )
       {
              printf("FATAL ERROR: Last error 0x%x , Ending program.\n " ,
GetLastError()  );
              return(1);
       }
       for ( i = 1   ;   i < max_clients + 1   ;   i++   )
       {
              (*ptr_threads)[i] = NULL ;
              (*ptr_args)[i].junctions = junctions;
              (*ptr_args)[i].mutex_file = ptr_mutex_file;
              (*ptr_args)[i].mutex_graph = ptr_mutex_graph;
```

```
                (*ptr_args)[i].num_of_junctions = num_of_junctions;
                (*ptr_args)[i].output_file = output_file ;
                (*ptr_args)[i].ptr_graph_matrix = ptr_graph_matrix;
                (*ptr_args)[i].ptr_quit = ptr_quit;
                (*ptr_args)[i].ptr_failure = ptr_failure;
        }

        *ptr_mutex_graph = CreateMutex(NULL , 0 ,NULL);
        if ( *ptr_mutex_graph == NULL )
        {
                printf("FATAL ERROR: Last error 0x%x , Ending program.\n " ,
GetLastError()  );
                return(1);
        }

        *ptr_mutex_file = CreateMutex(NULL , 0 ,NULL);
        if ( *ptr_mutex_file == NULL )
        {
                printf("FATAL ERROR: Last error 0x%x , Ending program.\n " ,
GetLastError()  );
                return(1);
        }

        return (0);
}


int Find_First_Unused_Theard_Slot( HANDLE *threads , int max_clients )
{
        int i = 0 ;
        DWORD wait_res = 0;

        for ( i = 1   ;   i < max_clients + 1   ;   i++ )
        {
                if ( threads[i] == NULL )
                        return (i);
                else
                {
                        wait_res = WaitForSingleObject ( threads[i] , 0 );
                        if (wait_res == WAIT_OBJECT_0)
                        {
                                CloseHandle(threads[i]);
                                threads[i] = NULL ;
                                break;
                        }
                }
        }
        return (i);
}


//Prepare the data that the server should send the client at the beginning of
thier communication.
// It copies the graph matrix in order to release it as quickly as possible and
not holdint it in all the sending data process.
//(The memory allocation free is done after sending the data to client).
int Preparing_Send_Data ( Data_From_Server *ptr_data_from_server , int
num_of_junctions , Junction* junctions ,
                                             int** original_graph , HANDLE
*mutex_graph )
{
        int i = 0 , j = 0 ;
        DWORD res = 0;

        ptr_data_from_server->graph_matrix = (int**)malloc( num_of_junctions *
sizeof(int*) );
        if ( ptr_data_from_server->graph_matrix == NULL)
        {
                printf("FATAL ERROR : Memory allocation failed.\n");
                return(1);
        }
        for ( i = 0  ;  i < num_of_junctions  ; i++ )
        {
                 (ptr_data_from_server->graph_matrix)[i] = (int*)malloc
(num_of_junctions * sizeof(int) );
                if ( (ptr_data_from_server->graph_matrix)[i] == NULL )
                {
                        printf("FATAL ERROR : Memory allocation failed.\n");
                        return(1);
                }
        }

        res = WaitForSingleObject( *mutex_graph , INFINITE );
        if (res == WAIT_FAILED )
        {
                printf("FATAL ERROR: Last error 0x%x , Ending program.\n " ,
GetLastError()  );
                return(1);
        }

        for ( i = 0  ;  i < num_of_junctions  ;  i++ )
        {
                for ( j = 0  ;  j < num_of_junctions  ;  j++ )
                        (ptr_data_from_server->graph_matrix)[i][j] =
original_graph[i][j];
        }
        res = ReleaseMutex( *mutex_graph );
        if (res == 0 )
        {
                printf("FATAL ERROR: Last error 0x%x , Ending program.\n " ,
GetLastError()  );
                return(1);
        }

        ptr_data_from_server->num_of_junctions = num_of_junctions;
        ptr_data_from_server->junctions = junctions;
        return(0);
}


BOOL Check_For_Failed_Threads ( HANDLE* threads , int max_clients )
{
        int i = 0 ;
        DWORD   exit_code = 0 ;
        BOOL success = 0 ;
        for ( i = 1  ;  i < max_clients + 1   ;   i++ )
        {
                if (threads[i] != NULL )
                {
                        success = GetExitCodeThread ( threads[i] , &exit_code );
                        if ( exit_code == 1 || success == 0 )
                        {
                               if (success == 0)
                               {
                                      printf("FATAL ERROR: Last error 0x%x , Ending
program.\n " , GetLastError()  );
                                      return (TRUE);
                               }
                        }
                }
        }
        return(FALSE);
}


int Single_Client_Func ( Single_Thread_Arg   *arg )
{
        int retval = 0 , source = 0 , destination = 0 , new_weight = 0;
        DWORD res = 0 ;
        TransferResult_t send_res, recv_res;
        Data_From_Server data_from_server;
        Updated_Arc data_from_client = { {0,0} , {0,0} , 0 };

        retval = Print_Server_Mode ( SUCCESSFULLY_CONNECTED , arg->output_file ,
arg->mutex_file , arg->client_serial_number ,
                                                      data_from_client , 0 );
        if (retval == 1 )
        {
                *(arg->ptr_failure) = TRUE;
                closesocket ( arg->s);
                return(1);
        }

        retval = Preparing_Send_Data (&data_from_server ,arg->num_of_junctions ,
arg->junctions , *(arg->ptr_graph_matrix) , arg->mutex_graph);
        if(retval == 1)
        {
                *(arg->ptr_failure) = TRUE;
                closesocket ( arg->s);
                free (data_from_server.graph_matrix);
                return(1);
        }

        send_res = Server_Sending_Data ( data_from_server , arg->s );
        if ( send_res == SETUP_PROBLEM   ||   send_res == TRNS_FAILED )
        {
                *(arg->ptr_failure) = TRUE;
                if ( send_res == SETUP_PROBLEM)
                        printf("FATAL ERROR: Memory allocation failed.\n");
                else
                        printf("FATAL ERROR: Service socket error while writing,
closing thread.\n");
                closesocket ( arg->s );
                free (data_from_server.graph_matrix);
                return(1);
        }
        free (data_from_server.graph_matrix);

        retval = Print_Server_Mode ( GRAPH_SENT , arg->output_file , arg-
>mutex_file , arg->client_serial_number ,
                                                      data_from_client , 0 );
        if (retval == 1 )
        {
                *(arg->ptr_failure) = TRUE;
                closesocket ( arg->s);
                return(1);
        }

        while(  *(arg->ptr_quit) == FALSE   &&   *(arg->ptr_failure) == FALSE  )
        {
                recv_res = Server_Receiving_Data(  &data_from_client ,  arg->s  );
                if ( recv_res == SETUP_PROBLEM   ||   recv_res == TRNS_FAILED )
                {
                        *(arg->ptr_failure) = TRUE;
                        if ( recv_res == SETUP_PROBLEM)
                               printf("FATAL ERROR: Memory allocation failed.\n");
                        else
                               printf("FATAL ERROR: Service socket error while
reading, closing thread.\n");
                        closesocket (arg->s);
                        return(1);
                }
                if (recv_res == TRNS_DISCONNECTED)
                {
                        retval = Print_Server_Mode ( CLIENT_DICONNECTED , arg-
>output_file , arg->mutex_file , arg->client_serial_number ,
                                                      data_from_client , 0 );
                        if (retval == 1 )
                        {
                               *(arg->ptr_failure) = TRUE;
                               closesocket (arg->s);
                               return(1);
                        }

                        retval = closesocket ( arg->s);
                        if ( retval == SOCKET_ERROR )
                        {
                               *(arg->ptr_failure) = TRUE;
                               printf("Error at closesocket(): %ld.\n" ,
WSAGetLastError() );
                               return(1);
                        }
                        return(0);
                }

                Check_If_Junction_Exist (arg->junctions , data_from_client.source ,
data_from_client.destination , &source , &destination , arg->num_of_junctions );
                res = WaitForSingleObject ( *(arg->mutex_graph) , INFINITE);
                if (res == WAIT_FAILED )
                {
                        *(arg->ptr_failure) = TRUE;
                        printf("FATAL ERROR: Last error 0x%x , Ending program.\n " , 
GetLastError()  );
                        closesocket ( arg->s);
                        return(1);
                }

                new_weight = ceil( 0.75 * (  *(arg->ptr_graph_matrix)
)[source][destination] +
                                             0.25 * data_from_client.delay  );

                (  *(arg->ptr_graph_matrix)  )[source][destination] = new_weight;
                (  *(arg->ptr_graph_matrix)  )[destination][source] = new_weight;
                res = ReleaseMutex( *(arg->mutex_graph) );
                if (res == 0 )
                {
                        *(arg->ptr_failure) = TRUE;
                        printf("FATAL ERROR: Last error 0x%x , Ending program.\n " , 
GetLastError()  );
                        closesocket ( arg->s);
                        return(1);
                }
                retval = Print_Server_Mode ( UPDATED_ARC , arg->output_file , arg-
>mutex_file , arg->client_serial_number ,
                                                      data_from_client , new_weight 
);
                if (retval == 1 )
                {
                        *(arg->ptr_failure) = TRUE;
                        closesocket ( arg->s);
                        return(1);
                }
        }

        retval = closesocket ( arg->s);
        if ( *(arg->ptr_failure) == TRUE )
                return(1);
        else
        {
                if ( retval == SOCKET_ERROR )
                {
                        *(arg->ptr_failure) = TRUE;
                        printf("Error at closesocket(): %ld.\n" , WSAGetLastError() );
                        return(1);
                }
                return(0);
        }

}
```