CSCE 313-200: Computer Systems

Homework #1 (100 pts)

Due dates: 1/25/18 (part 1), 2/8/18 (part 2), 2/22/18 (part 3)

1. Purpose

Understand the Visual Studio environment, creation of projects, simple process debugging, search algorithms, and basic inter-process communication.

2. Problem Description (Part 1)

Your goal is to implement a number of multi-threaded search algorithms on weighted graphs being held by another process in the system. This conveniently maps to the following problem that might be easier to understand. Assume that your job is to navigate a space rover out of a cave located on some remote planet. Each cave consists of a maze of (mostly) dark rooms interconnected by tunnels, all of which can be represented by a giant undirected graph (i.e., each room is a node, each tunnel between rooms is an edge).

Since the rover is slow, it cannot search for the exit directly. However, it can unleash *N* search flybots to explore the cave. Due to their small size, these robots are relatively dumb, which means that they can neither remember which rooms they have been to nor coordinate with each other to avoid covering the same room multiple times. Each flybot is equipped with a wireless link that allows it to receive navigation directions from the rover (i.e., which room to explore next). The result of each visit is the list of neighboring rooms and the amount of light visible to the flybot in each of them, which are transmitted back to the rover to aid the search. The high-level objective is to concurrently control the robots so as to find the unique exit in the shortest amount of time.

The topology of the cave is unknown a-priori and must be explored in real-time. The rover contains a module called *Command Center* (CC), which relays directives from your software to the robots and forwards their responses back to you. See Figure 1 for an illustration. The CC process and its standardized control directives are provided as part of this homework. During execution, you must first launch the CC and then communicate with it for all subsequent navigation of the cave through a message-based channel (or multiple channels).

It should be noted that responses from flybots are not instantaneous since there is some inherent delay needed to move from one room to another. This results not only in large, but also randomly fluctuating, response delays. This puts severe constraints on how much exploration can be done with a single robot. However, with multi-threading you should be able to successfully escape caves with 10^7 nodes.

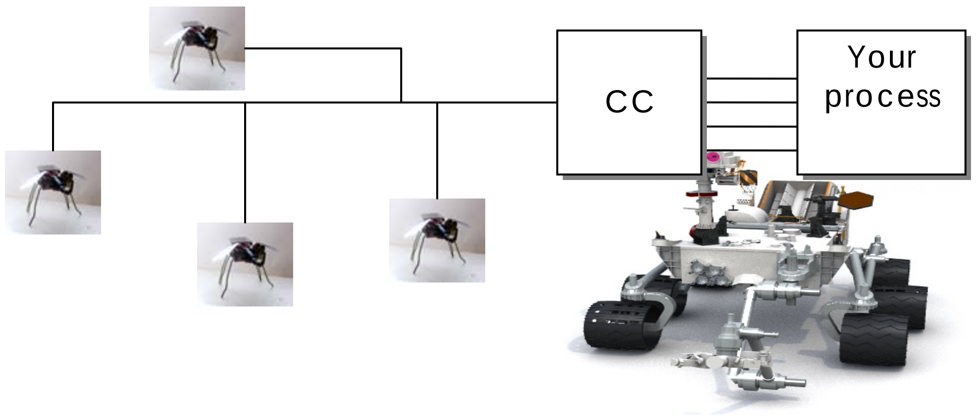


Figure 1. Operation of the CC with 4 flybots.

During the initial connection to the CC, you must specify a particular planet *P* and cave *C* within that planet, where $1 \leq P \leq 7$ determines how large the cave is (size = 10^P rooms) and $C \geq 0$ specifies which random instance of that cave you will attempt to navigate. For example, tuple (3, 7) refers to the 7th random instance of some cave with 1,000 rooms. This notation allows you to test your code with small caves to start with (i.e., P = 1 or 2) and then transition to larger ones (i.e., P = 7) as your program becomes more efficient. In addition, this model allows you to replay old scenarios (i.e., by specifying the same *C*), while having a virtually unlimited supply of new caves to experiment with (i.e., by varying *C*).

2.1. Code (25 pts)

The program must accept two command-line arguments that identify the planet and cave to be explored:

```
caveSearch.exe 6 44
```

where this example means planet 6, cave 44. Your code should check if the correct parameters are present and otherwise print usage info.

You must be able to connect to the CC, print its response, then open a new channel to the first robot, issue another connect, and print the initial room ID from that response. Then, your program must cleanly shut down the robot, then the CC, and finally terminate after the CC process exits. Your printouts should be of this form:

```
Starting CC.exe...
*** CC: ver 1.5 starting with PID 24700 (hex 607C)
*** CC: found 6 CPUs, 14220.61 MB of free RAM
Connecting to CC with planet 4, cave 4, robots 1...
*** CC: creating graph (1st pass)
*** CC: creating graph (2nd pass)
*** CC: building a map
*** CC: propagating light sources
*** CC: finished initialization
CC says: status = 1, msg = ‘opened (planet 4, cave 4) with 1 robot(s)’
Connecting to robot 0...
Robot says: status = 1, msg = ‘connected’
Current position: room 8DCD0AE66BDA38D9, light intensity 0.00
-------------------------
Disconnecting robot 0...
```

```
    Disconnecting CC...
    Waiting for CC.exe to quit...
    *** CC: main thread waiting for robots...
    *** CC: all robots finished
    *** CC: quitting, kernel time 0.00 sec, user time 0.01 sec
    Execution time 1.77 seconds
```

Note that every line that starts with *** comes from CC.exe, while everything else is printed by your program. Also observe that the execution time must be displayed only *after* CC.exe has quit.

Efficient coding and well-structured programming is expected. You may lose points for copy-pasting the same function (with minor changes) over and over again, for writing poorly designed or convoluted code, *not checking for errors in every single API you call*, and allowing buffer overflows, access violations, debug-assertion failures, heap corruption, synchronization bugs, memory leaks, or conditions that lead to a crash. Furthermore, your program must be robust against unexpected responses from the CC and deadlocks.

2.2. Process Creation

There are two versions of the CC process – 32 and 64 bit. The former uses slightly less memory, but is limited in the number of threads it can start (about 5,000). For more massive runs in Part 3, use the 64-bit version.

Creating a process is rather simple:

```
PROCESS_INFORMATION pi;
STARTUPINFO s;

GetStartupInfo (&s);

char path [] = "CC.exe";
if (CreateProcess (path, NULL, NULL, NULL,
        false, 0, NULL, NULL, &s, &pi) == FALSE)
{
        printf (“Error %d starting CC\n”, GetLastError());
        exit(-1);
}
```

On return, structure `pi` contains two important fields – the *handle* of the created process and its *process ID* (PID). Using the former, you will need to pause at the end until the CC process terminates (see `WaitForSingleObject`). Using the latter, you will open a named pipe and communicate with the CC as described next.

2.3. Named Pipes

This homework explores an inter-process communication primitive called *named pipe*, which is a bidirectional, lossless, synchronous channel between two processes. The first task in opening a pipe is to construct its name. To prevent multiple instances of the CC from interfering with each other on the same host, each pipe name must carry the hex PID of the CC. For example, `\\.\pipe\CC-17A5` is the name of a pipe that can be opened to a CC process with PID 17A5. You can use `sprintf` or STL string functions to create this name. Note that the PID comes from `CreateProcess` in Section 2.2.

There are two general steps needed to open a pipe once its name is known:

```
// wait for the CC to initialize the pipe from its end
while (WaitNamedPipe (pipename, INFINITE) == FALSE)
       Sleep (100);                  // pause for 100 ms

// now open the pipe
HANDLE CC = CreateFile (pipename,  GENERIC_READ | GENERIC_WRITE,
        0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
```

If creation is successful, the pipe can be read from and written to using `ReadFile` and `Write-File`. To close the pipe or any other open handle (e.g., mutex, semaphore), use `CloseHandle`.

2.4. CC Message Format

Make sure to pack all structs below to 1 byte, i.e., use `#pragma pack(push,1)` before declaring them and `#pragma pack(pop)` afterwards. After establishing a pipe, you can send messages to the CC, each consisting of the following 7 bytes:

```
class CommandCC {
public:
       uchar                 command:2;     // lower 2 bits
       uchar                 planet:6;      // remaining 6 bits
       DWORD                 cave;          // which cave
       ushort                robots;        // how many robots
};
```

Here, `command` can be either CONNECT = 0 or DISCONNECT = 1. The next two parameters (planet, cave) were provided earlier by the user. Set the number of robots to one. The CON-NECT command elicits the following response:

```
class ResponseCC {
public:
       DWORD                 status;
       char                  msg [64];
};
```

The status code can be either FAILURE = 0 or SUCCESS = 1, while `msg` is a NULL-terminated char buffer providing a text response to the command. You can directly use `printf` on the mes-sage. The DISCONNECT command requests that the CC shut down, which it does after waiting for all flybot threads quit. There is no response to DISCONNECT.

2.5. Robot Message Format

Once the CC is aware of the number of desired robots, it will start *N* internal threads and create a new pipe for each of them. For a given CC pipe `\\.\pipe\CC-17A5`, robot pipes will have names `\\.\pipe\CC-17A5-robot-0`, `\\.\pipe\CC-17A5-robot-1`, and so on up to *N*–1, where *N* was requested in the initial CONNECT message. Note that numbers 0, 1, 2, …, *N*–1 must be written in *hex*. Use `_itoa` or `sprintf` to accomplish this conversion. For example:

```
char robotPipeName [1024];
int k = 5;            // 5th robot
sprintf (robotPipeName, "\\\\.\\pipe\\CC-%X-robot-%X", PID, k);
```

Next, you will need to sleep-spin on `WaitNamedPipe` until these pipes become available and af-terwards issue `CreateFile` on each of them as explained in Section 2.3.

Robot requests have the following format:

```
class CommandRobot {
```

```
public:
       DWORD                  command;
       uint64                 room;          // unsigned __int64
};
```

where the command can be CONNECT = 0, DISCONNECT = 1, or MOVE = 2. For the first two commands, the room parameter is ignored (although it still must be provided in the message). For the last command, the room specifies where the robot should fly to. All rooms are random 8-byte hashes, i.e., they are *not* sequential from 0 to 2^P-1.

Robot responses start with a header that follows this structure:

```
class ResponseRobot {
public:
       DWORD                  status;
       char                   msg [64];
};
```

The `status` and `msg` fields are identical to those in CC responses. If the command is successful, the `ResponseRobot` header is followed by a list of neighboring room IDs (`uint64`) and light intensities (`float`). Starting with Part 2, the total length of each response can be determined using `PeekNamedPipe` (see Section 3.2). Upon failure, the message just stops at the header.

After the entire message is obtained from the OS via `ReadFile`, you can use a pointer of type `NodeTuple64` to read the list of neighbors:

```
class NodeTuple64 {
public:
       uint64                 node;
       float                  intensity;
};
```

The CONNECT command returns only one room in the response, which is where the rover is located. This room is the starting point of your search in Part 2. The DISCONNECT command shuts down the robot and does not elicit any response. It should be noted that the proper shutdown sequence is to first stop all robots, close their *N* pipes, then stop the CC, close its pipe, and finally wait for the CC process to quit.

Avoid hardwiring constants; instead, apply `sizeof()` to every class and use the following definitions:

```
#define CONNECT                          0
#define DISCONNECT                       1
#define MOVE                             2

#define FAILURE                          0
#define SUCCESS                          1
```

2.6. Debug Printouts

One useful technique is to print raw buffers byte-by-byte when you are unsure what is going on. For example, suppose `bufSize` bytes have just been received from the pipe into `someBuffer`. Then, you can dump the entire buffer to screen using a simple loop:

```
char *someBuffer;
for (int i = 0; i < bufSize; i++)
       printf ("%X ", someBuffer [i]);
```

For more information on `printf`, see:

http://msdn.microsoft.com/en-us/library/56e442dc(v=vs.71).aspx

2.7. Traces

Sample run:

```
Starting CC.exe...
*** CC: ver 1.5 starting with PID 24700 (hex 607C)
*** CC: found 6 CPUs, 14220.61 MB of free RAM
Connecting to CC with planet 1, cave 1, robots 1...
*** CC: creating graph (1st pass)
*** CC: creating graph (2nd pass)
*** CC: building a map
*** CC: propagating light sources
*** CC: finished initialization
CC says: status = 1, msg = ‘opened (planet 1, cave 1) with 1 robot(s)’
Connecting to robot 0...
Robot says: status = 1, msg = ‘connected’
Current position: room E8825E744AF37DEB, light intensity 2.85
-------------------------
Disconnecting robot 0...
Disconnecting CC...
Waiting for CC.exe to quit...
*** CC: main thread waiting for robots...
*** CC: all robots finished
*** CC: quitting, kernel time 0.00 sec, user time 0.07 sec
Execution time 1.12 seconds
```

A few more initial positions:

```
planet 5, cave 15: room 2346CD5D02A2D914, light intensity 0.00
planet 6, cave 1300: room 1971C9F35798C4B2, light intensity 0.00
planet 7, cave 320: room 72E191306EAD0A94, light intensity 0.00
```

The next example shows what happens with an invalid planet. Your program must not continue when it encounters errors, but it may terminate without cleanup in fatal cases. As shown below, unclean shutdown forces the CC to throw an error shown in bold (since the kernel closes the pipe after your process quits), which is normal:

```
Starting CC.exe...
*** CC: ver 1.5 starting with PID 24700 (hex 607C)
*** CC: found 6 CPUs, 14220.61 MB of free RAM
Connecting to CC with planet 50, cave 1, robots 1...
CC says: status = 0, msg = ‘connect error: only planets 1-7 are supported’
Connection error, quitting...
*** CC: error 109 reading the pipe, exiting
```

3. Problem Description (Part 2)

In this part, you will perform a single-threaded search using four different techniques.

3.1. Code (25 pts)

Your program must now accept three command-line arguments: planet, cave, and the search method:

```
caveSearch.exe 6 44 BFS
```

The other search options are DFS, bFS, and Astar, where all strings are case-sensitive. The print-outs should contain all of those in Part 1 and additionally the following:

```
    Initial position: room 8DCD0AE66BDA38D9, light intensity 0.00
    Starting search using BFS...
```

```
1) visiting 8DCD0AE66BDA38D9, degree 1, discovered 2
2) Visiting 75EB693C57F16B20, degree 28, discovered 29
3) Visiting C14E6529DAD59EC5, degree 40, discovered 68
4) Visiting 27230CD7014A1D42, degree 7, discovered 74
5) Visiting 64ED8B5283AFFD6B, degree 9, discovered 82
...
8959) visiting F9C145893D12537A, degree 3, discovered 9959
Found exit A9E402E1282F881, 8960 steps, distance 6
```

For every visited room, print the current search step (starting with the initial room where the rover is positioned), room ID, its degree (i.e., number of neighbors), and the total number of unique rooms discovered thus far after processing the current room.

During the search, you cannot allow multiple visits to the same room, which means that you must run duplicate elimination on the discovered rooms. Furthermore, you are not allowed to deploy any *linear-scan* algorithms to maintain this data structure. In fact, you are strictly limited to $O(\log n)$ complexity per found room, where n is the cave size.

3.2. Reading Messages of Unknown Size

Note that robot responses contain an unknown amount of data in each message. Furthermore, this data may not fit into your a-priori allocated buffer. The recommended course of action is to start with a fixed buffer size $B = 128$. After the first call to `ReadFile`, if you observe that the buffer is filled to the limit, you can issue `PeekNamedPipe` to retrieve the amount of additional data still stuck in the pipe. Then, you can resize the initial buffer using `realloc` (or `HeapReAlloc`) and read the second part of the response into an appropriate offset within that buffer. If you prefer to use `new/delete`, you can allocate a new buffer, copy the first received portion into it (i.e., B bytes), delete the old buffer, and issue the second `ReadFile` to consume the remaining data.

Note that when the message is the exact size as the buffer, you will see 0 remaining bytes when peeking at the pipe. In such cases, *it is crucial that you avoid trying to read these 0 bytes* from the pipe as the kernel will wait for something more to be sent from the CC, which will deadlock your code.

3.3. Exit

When your search issues a MOVE to the exit room, the robot returns a SUCCESS response with no neighbors (i.e., just the header).

3.4. Light Dynamics

The exit and certain other rooms (e.g., with holes in the ceiling) are the only sources of light in the otherwise dark cave. Each planet obeys simple rules of physics: the light gradually fades the further you move from its source. Figure 2 shows one example of light propagation with an exponential decay. In this example, the light is *additive*, meaning that incoming brightness from multiple sources is accumulated; however, the exact dynamics of light propagation are not essential to finding the exit.

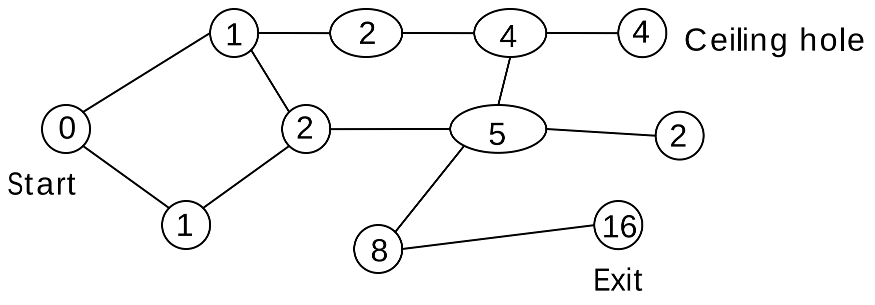


Figure 2. Sample cave (numbers indicate brightness of the room).

3.5. Search Techniques

The studied algorithms – BFS, DFS, bFS, and A* – are explained in more detail below. In all four cases, your program should maintain 1) a set *U* of *unexplored* rooms, which is built dynamically from MOVE responses; and 2) a set *D* of *discovered* rooms, which you will use to prevent infinite looping.

In BFS, recall that *U* is simply a FIFO queue. In DFS, it is a LIFO stack. For bFS, we leverage the observation that brighter rooms are more likely to lead toward the exit and should be preferred compared to darker rooms. Thus, at each step, your program can extract from *U* the *next brightest* room and attempt to navigate it. While this does not guarantee that you will find the globally optimal (i.e., shortest) path, it ensures that you are likely to get to it quickly. In the worst case, bFS covers the entire graph and thus has the same upper bound on complexity as the two traditional methods.

A* search is similar to bFS, but it additionally takes into account the *distance* from the starting position. Assume each unexplored node i has quality $Q_i = L_i + w / (d_i + 1)$, where L_i is the light intensity of the room, d_i is its distance from the rover, and w is some weight that assigns preference between bFS (i.e., small w) or BFS (i.e., large w). In fact, $w = 0$ produces bFS and $w = \infty$ makes BFS. Unless you find a better constant, use $w = 20$ in this homework. Nodes are then sorted according to their quality and the best one is explored first.

3.6. Implementing Search

Note that your main loop of the search algorithm must be written once (i.e., without knowing the underlying implementation for *U* or *D*). A good starting approach is to define a base class for *U* with all functions (e.g., `push, pop, size`) declared as pure virtual:

```
class UnexploredRoom {                              // class returned from pop()
    uint64                 ID;                      // room ID
    int                    distance;                // distance from source
};

class Ubase {
        ...
    virtual void           push (uint64 roomID,     // pushes the next room into U
                                   float intensity, int distance) = 0;
    virtual UnexploredRoom pop (void) = 0;          // pops the next room
    virtual int               size (void) = 0;      // checks the size of U
        ...
};
```

The next step is to inherit from `Ubase` four separate classes `Ubreadth, Udepth, Ubest, Uastar`, each using a different algorithm for managing *U* and overwriting the virtual functions. The first two use respectively an STL queue and stack that contain pairs (room ID, distance). The other two classes use priority queues that store triples (quality, room ID, distance). These should be sorted in descending order of quality. For bFS, the quality is just the light intensity of the room. For A*, it is Q_i given in Section 3.5.

A similar encapsulating technique applies to the discovered set *D*, except you won't have to inherit anything from it since there is only one type of *D* in this homework (i.e., an STL set):

```
class Discovered {
        ...
        bool CheckAdd (uint64 roomID);       // add to set if not there already
};
```

Then, instantiating a BFS search would look something like this:

```
Discovered d;
NodeTuple64 *nt = ...                  // get initial room pointer from robot

Ubreadth ub;
Search (&ub, &d, nt->room);            // provide U, D, and initial room
```

and that for DFS:

```
Udepth ud;
Search (&ud, &d, nt->room);
```

Note that the same functionality can be accomplished in C using *function pointers* (i.e., without inheritance), but the C++ version above is probably easier to manage and debug.

3.7. Robot Responses

For Parts 2-3, a maximum of *two* read calls are allowed. After up-sizing the buffer, you may want to retain it for the next read operation. After reaching some critical size, the buffer will stabilize and allow receipt of all future messages in one call to ReadFile. The only exception to this algorithm is the extra-credit portion in Part 3 where the monster will aim to make your code exhaust system RAM and crash. The monster does this by dumping huge messages into the pipe. In such cases, the buffer should be kept only if it is less than 5 KB.

3.8. Traces

This run is from planet 2, cave 1 (initial and final exchanges omitted):

```
Starting search using BFS...
1) Visiting 60C2F3EFD35E9B6B, degree 1, discovered 2
2) Visiting 443A9EAC822367E1, degree 8, discovered 8
3) Visiting 9823C27639428949, degree 37, discovered 38
4) Visiting F98B230317993E18, degree 14, discovered 45
5) Visiting 180ECE05813FBC81, degree 8, discovered 47
...
76) Visiting 756653282582D497, degree 2, discovered 99
77) Visiting 9072D980F494CBEC, degree 4, discovered 99
78) Visiting B20157647C43F450, degree 3, discovered 99
```

```
79) Visiting A6127CA5C3DB82C3, degree 1, discovered 99
Found exit DE86B6F20F218E9F, 80 steps, distance 4

Starting search using DFS...
1) Visiting 60C2F3EFD35E9B6B, degree 1, discovered 2
2) Visiting 443A9EAC822367E1, degree 8, discovered 8
3) Visiting B8290D5B1751A30D, degree 3, discovered 10
4) Visiting 828340DCF134BEFE, degree 6, discovered 14
5) Visiting A856FA6A866C30E0, degree 1, discovered 14
...
17) Visiting 78E549C629E1641A, degree 4, discovered 38
18) Visiting 8C6AB86D5C741C94, degree 2, discovered 38
19) Visiting 3B44B813AEFFE593, degree 3, discovered 38
Found exit DE86B6F20F218E9F, 20 steps, distance 13

Starting search using bFS...
1) Visiting 60C2F3EFD35E9B6B, degree 1, discovered 2
2) Visiting 443A9EAC822367E1, degree 8, discovered 8
3) Visiting F98B230317993E18, degree 14, discovered 19
...
7) Visiting FC395132A65451BE, degree 8, discovered 39
8) Visiting E8825E744AF37DEB, degree 6, discovered 40
Found exit DE86B6F20F218E9F, 9 steps, distance 6

Starting search using A*...
1) Visiting 60C2F3EFD35E9B6B, degree 1, discovered 2
2) Visiting 443A9EAC822367E1, degree 8, discovered 8
3) Visiting F98B230317993E18, degree 14, discovered 19
...
9) Visiting FC395132A65451BE, degree 8, discovered 63
10) Visiting E8825E744AF37DEB, degree 6, discovered 64
Found exit DE86B6F20F218E9F, 11 steps, distance 5
```

4. Problem Description (Part 3)

In this part, you will multi-thread the previous version of the homework and analyze search results by answering a number of questions posed below.

4.1. Code (25 pts)

Your program must now accept four command-line arguments: planet, cave, number of threads to run, and the search method:

```
caveSearch.exe 6 44 300 BFS
```

First, make sure your program prints the initial CC exchange as in part 1, but omits the robot responses. Second, a dedicated stats thread must print in the following format every 2 seconds:

```
    [2s] E 10.79K, U 23.83K, D 34.62K, 5362/sec, active 4894, run 5000
```

which shows that 2 seconds have elapsed, 10.79K rooms have been sent to robots for exploration (i.e., removed from *U*), 23.83K additional rooms are still pending in *U*, 34.62K rooms have been discovered, the current exploration rate is 5,362 rps (rooms/sec), there are 4,894 active threads with a room to explore, and a total of 5,000 threads are still running. Note that the first four values are *cumulative* (i.e., from the start time), while the exploration rate is computed since the last printout. The thread counts in the last two columns are *instantaneous*, i.e., taken when the printout is made.

When a search thread finds the exit, it must notify all others to stop searching and print its thread ID (in the range 0 to *N*–1), exit room number, how many rooms have been explored so far across all robots, and the distance along the discovered path:

```
Thread 9223: found exit AABF6237BFD5B685, steps 386416, distance 8
```

After all threads have shut down their robots and quit, the final printout summarizes the run. Note that the corresponding crawling rate is computed cumulatively since the beginning.

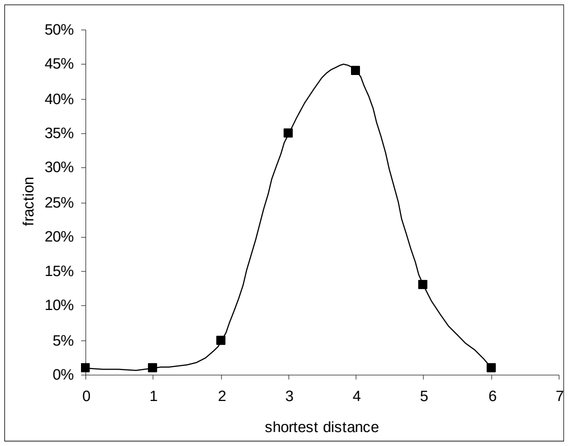
```
[final] E 413.90K, U 467.63K, D 881.53K, 12968/sec, active 0, run 0
Execution time: 34.66 seconds
```

It is not advisable to wake up threads if there is no work for them to do or constantly create/terminate them during the run (i.e., all *N* threads must be started at the beginning and kept going until the exit is found). Use the producer-consumer algorithm from class.

4.2. Report (25 pts)

It is recommended to allocate several days to analyze the data, run the experiments, and write about your results. Breakdown of points:

1. (5 pts) Disable the exit (i.e., search the entire graph) and experiment with BFS on planet 7, cave 55 by increasing the number of threads from 1 to 15,000 and document performance improvements arising from parallelization of the search. Specifically, plot the maximum stable search rate (e.g., after 120 seconds) vs. the number of threads and use curve-fitting to see how well this approximates a linear function. Using the slope of this curve, determine the average delay needed for a flybot to satisfy your request.
2. (5 pts) Use a *single* thread on planet 3, compute the *average* number of steps needed by each of the search methods (BFS, DFS, bFs, and A*) to find the exit in caves 40-49. Comment on the number of visited rooms by each method and the quality of the solution found (i.e., distance from the starting room). Does any one method win decisively?
3. (10 pts) Using 10K threads, plot a distribution of *shortest* distances (obtained using BFS) from the rover to each of the available rooms in cave 944 on planet *P* = 6. By eyeballing the figure, comment on whether this distribution can be approximated as Gaussian. To make results meaningful, this should be plotted using a *normalized* histogram (also known as the *probability mass function*) where the *y*-axis contains the *percentage*, rather than the count, of rooms whose distance falls into a given bin on the *x*-axis. Figure 3(a) shows the result for planet 2; you will need a similar plot for *P* = 6. See http://en.wikipedia.org/wiki/Histogram and http://en.wikipedia.org/wiki/Probability_mass_function for more details.
4. (5 pts) Using 5K threads, study cave *C* = 60 on planets *P* = 2-7 and examine how the average *shortest* distance from the rover to all other rooms scales with planet size. It was observed that many random graphs asymptotically grow average distance as $\log_b(n)$, where *n* is the total number of nodes and *b* is a constant that depends on the average degree. Prove or disprove this conjecture and then experimentally determine *b* using curve fitting.



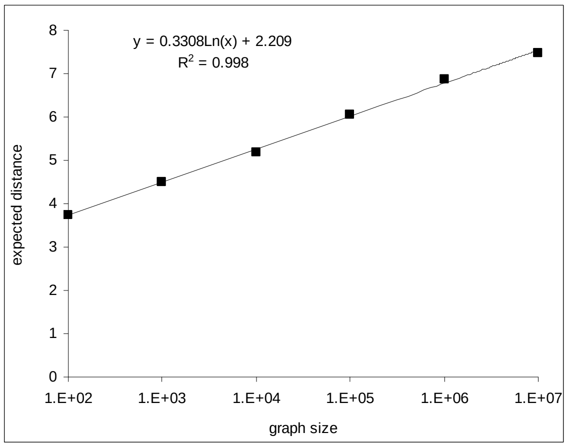


Figure 3. (a) Distribution of shortest distances for *P* = 2, *C* = 44. (b) Average distance for *C* = 40.

4.3. Extra Credit (20 pts)

Special caves on planets 6-7 are populated with a monster whose goal is to impede your escape. The main goal with this extra-credit assignment is to use overlapped I/O to overcome deadlocks and various message corruption to successfully navigate monster caves. All caves numbered 1,000 and higher have a monster in them.

The monster interferes in several ways – by eating a flybot, corrupting the response header, truncating the message, padding the room list, and crashing the process when you direct flybots to an invalid room. An eaten flybot results in either a timeout on `WriteFile`/`ReadFile` or an error in Windows APIs related to the pipe (e.g., `ReadFile`, `WriteFile`, `PeekNamedPipe`, `ConnectNamedPipe`, `WaitNamedPipe`, `WaitForSingleObject`, `GetOverlappedResult`). You must catch these conditions, place the room where the robot was killed back to the unexplored set *U*, and quit your thread that corresponds to this robot.

For jammed transmissions, the monster may corrupt the status code to become invalid (i.e., neither SUCCESS nor FAILURE) or truncate the message to a size smaller than the minimum valid length. It may also add bogus room numbers to the message in an attempt to confuse your search and ultimately cause the CC to crash when you attempt to navigate to an invalid room. However, when padding messages, the monster always produces a list in which the last `NodeTuple64` is incomplete (i.e., less than 12 bytes), which allows you to easily detect corrupted responses. For both jammed and padded messages, the correct course of action is to discard them and keep retrying the room in the same thread until success.

To help debug threads hanging in the CC, you can specify a command-line argument 'debug' to CC.exe, which will cause it to periodically print the status of its threads:

```
CC says: status = 1, msg = 'opened (planet 6, cave 1001) with 15000 robot(s)'
*** CC status: 1 (14963)  2 (32)  4 (5)
*** CC status: 1 (14479)  2 (85)  4 (436)
[2s] E 1.69K, U 15.38K, D 17.06K, 838/sec, active 876, run 879
*** CC status: 1 (14004)  2 (72)  4 (923)  7 (1)
*** CC status: 1 (13722)  2 (57)  4 (1218)  7 (2)  8 (1)
[4s] E 6.02K, U 52.15K, D 58.18K, 2146/sec, active 1498, run 1503
*** CC status: 1 (13315)  2 (116)  4 (1565)  5 (1)  7 (2)  8 (1)
*** CC status: 1 (12615)  2 (223)  4 (2155)  5 (3)  7 (1)  8 (3)
```

```
[6s] E 14.15K, U 107.82K, D 121.97K, 4038/sec, active 2961, run 2966
*** CC status: 1 (11594)  2 (292)  4 (3101)  5 (5)  7 (1)  8 (7)
*** CC status: 1 (9654)  2 (383)  4 (4946)  5 (2)  7 (1)  8 (14)
[8s] E 35.42K, U 183.32K, D 218.75K, 10565/sec, active 9540, run 9622
*** CC status: 2 (3154)  4 (11815)  7 (2)  8 (29)
*** CC status: 4 (14945)  7 (4)  8 (51)
[10s] E 85.47K, U 334.36K, D 419.83K, 24873/sec, active 14893, run 14953
*** CC status: 4 (14918)  7 (5)  8 (77)
```

The printouts show the number of threads (in parentheses) in each of the states 0 through 8. State 0 indicates the robot has not started yet, 1 is waiting for the pipe to be created in your thread, 2 waiting for the connect message, 4 expecting a MOVE command, 5 navigating to a room, 7 sending back the neighbors, and 8 shutting down after a DISCONNECT or consumption by the monster. Additional codes 3 and 6 indicate error conditions related to invalid initial commands and insufficient length of messages, but these should not arise if your program successfully completes non-monster caves.

For example, the last line of the above run shows that 14.9K threads are waiting for the MOVE command, 5 are about to send a reply, and 77 have been eaten by the monster. This debug information may become helpful when some of the robots never quit and hang your threads indefinitely. Knowing what they are doing should aid in ensuring a clean shutdown.

4.4. Caveats

It is crucial that you put robot pipe creation and initial connections into separate threads rather than attempt to run them from `main()`. Otherwise, you will wait a loooooong time for the initialization to complete in the main thread.

When benchmarking with a large number of threads, *always run your code in release mode*, where STL is 50 times faster than in debug mode and occupies 50% less memory. STL is also notoriously slow in releasing memory when compiled in debug mode, which sometimes creates an illusion of a deadlock when your code frees a large chunk of RAM just before terminating.

To avoid swapping to disk and showing low performance in your report, check that the total memory usage in Task Manager is well below your physical RAM size. You can notice that something is wrong when increasing the number of threads beyond some threshold (such as 2,500) leads to lower performance. Keep in mind that under Win32, each process is limited to 2 GB of RAM, which caps the number of threads to about 5-6K; however, even this requires changing the default thread stack size from 1 MB to 64 KB (see Project Properties → Linker → System → Stack Reserve & Commit Size). Without this modification, the maximum number of Win32 threads you can run is about 1,400. Therefore, it is recommended to always compile your code in x64.

4.5. Traces

Observe below that there is a noticeable delay in getting all threads to start and that at any given time about 20-50 robots are inactive (i.e., not exploring any rooms):

```
CC says: status = 1, msg = 'opened (planet 6, cave 15) with 15000 robot(s)'
Starting search using BFS...
[2s] E 0.00K, U 0.00K, D 0.00K, 1/sec, active 1, run 675
[4s] E 3.65K, U 31.17K, D 34.82K, 1812/sec, active 1411, run 1412
[6s] E 10.90K, U 87.07K, D 97.97K, 3600/sec, active 2489, run 2498
```

```
[8s] E 22.02K, U 152.73K, D 174.75K, 5526/sec, active 3891, run 3896
[10s] E 39.08K, U 224.62K, D 263.70K, 8474/sec, active 5701, run 5706
[12s] E 64.28K, U 301.83K, D 366.11K, 12524/sec, active 8260, run 8270
[14s] E 98.49K, U 372.63K, D 471.12K, 17002/sec, active 11218, run 11249
[16s] E 143.53K, U 432.16K, D 575.68K, 22373/sec, active 14494, run 14527
[18s] E 193.05K, U 472.75K, D 665.80K, 24614/sec, active 14981, run 15000
[20s] E 242.88K, U 490.74K, D 733.62K, 24751/sec, active 14952, run 15000
[22s] E 292.81K, U 493.04K, D 785.85K, 24811/sec, active 14959, run 15000
[24s] E 342.52K, U 484.69K, D 827.21K, 24702/sec, active 14955, run 15000
Thread 9223: found exit AABF6237BFD5B685, steps 386416, distance 8
[27s] E 386.52K, U 479.06K, D 865.57K, 11603/sec, active 0, run 0
[final] E 386.52K, U 479.06K, D 865.57K, 13834/sec, active 0, run 0
Waiting for CC.exe to quit...
*** CC: main thread waiting for robots...
*** CC: all robots finished
*** CC: quitting, kernel time 0.14 sec, user time 0.18 sec
*** Main user: quitting, kernel time 0.70 sec, user time 0.06 sec
Execution time: 34.66 seconds
```

Another example are caves 900-999 that do not have an exit:

```
CC says: status = 1, msg = 'opened (planet 6, cave 950) with 9000 robot(s)'
Starting search using BFS...
[2s] E 4.79K, U 5.13K, D 9.92K, 2383/sec, active 4213, run 4230
[4s] E 31.01K, U 166.70K, D 197.72K, 13023/sec, active 8968, run 9000
[6s] E 60.96K, U 286.30K, D 347.26K, 14887/sec, active 8971, run 9000
[8s] E 90.92K, U 361.51K, D 452.43K, 14888/sec, active 8973, run 9000
...
[66s] E 955.71K, U 42.20K, D 997.91K, 14884/sec, active 8965, run 9000
[68s] E 985.74K, U 13.56K, D 999.30K, 14909/sec, active 8967, run 9000
[70s] E 999.99K, U 0.00K, D 999.99K, 7068/sec, active 220, run 9000
Thread 4837: reached empty queue!!
[final] E 1000.00K, U 0.00K, D 1000.00K, 13975/sec, active 0, run 8772
Waiting for CC.exe to quit...
*** CC: main thread waiting for robots...
*** CC: all robots finished
*** CC: quitting, kernel time 0.12 sec, user time 0.22 sec
*** Main user: quitting, kernel time 0.74 sec, user time 0.09 sec
Execution time: 75.22 seconds
```

4.6. Monster Caves

The most interesting aspect here is that your code will lose threads at an average rate of one robot per 1,000 explored rooms. You will thus need at least 10K threads to finish planet 7, but 15-20K is recommended to keep the speed reasonable towards the end. You should also make sure that your program can handle the robot being killed *inside* the exit room as shown below.

```
Starting search using BFS...
CC says: status = 1, msg = 'opened (planet 6, cave 1001) with 15000 robot(s)'
[2s] E 1.43K, U 5.88K, D 7.31K, 710/sec, active 926, run 930
[4s] E 5.98K, U 43.30K, D 49.28K, 2256/sec, active 1669, run 1670
[6s] E 13.84K, U 102.92K, D 116.75K, 3906/sec, active 2677, run 2686
[8s] E 28.55K, U 165.92K, D 194.47K, 7313/sec, active 6652, run 6713
*** Monster: killing robot in exit room! -----------------------
[10s] E 92.36K, U 347.91K, D 440.26K, 24122/sec, active 14943, run 14949
*** Monster: killing robot in exit room! -----------------------
[12s] E 141.61K, U 428.57K, D 570.18K, 24466/sec, active 14880, run 14913
*** Monster: killing robot in exit room! -----------------------
[14s] E 190.63K, U 471.13K, D 661.76K, 24365/sec, active 14826, run 14870
[16s] E 239.81K, U 490.40K, D 730.21K, 24428/sec, active 14771, run 14814
*** Monster: killing robot in exit room! -----------------------
[18s] E 288.56K, U 493.34K, D 781.90K, 24227/sec, active 14708, run 14760
[20s] E 337.05K, U 485.99K, D 823.03K, 24094/sec, active 14653, run 14706
*** Monster: killing robot in exit room! -----------------------
[22s] E 385.22K, U 470.97K, D 856.20K, 23933/sec, active 14618, run 14656
*** Monster: killing robot in exit room! -----------------------
[24s] E 433.62K, U 449.66K, D 883.27K, 24048/sec, active 14565, run 14611
```

```
[26s] E 481.68K, U 423.61K, D 905.28K, 23870/sec, active 14527, run 14565
*** Monster: killing robot in exit room! ------------------------
[28s] E 529.82K, U 393.83K, D 923.65K, 23922/sec, active 14487, run 14518
Thread 5284: found exit 6DBA04BEDEC9B2A0, steps 567433, distance 6
[32s] E 567.51K, U 372.71K, D 940.22K, 10501/sec, active 8, run 8
[34s] E 567.51K, U 372.71K, D 940.22K, 0/sec, active 0, run 0
[final] E 567.51K, U 372.71K, D 940.22K, 16528/sec, active 0, run 0
```

313 Homework 1 Code (Part 1)

Name: ____________________

Function	Points	Break down	Item	Points
Basic code structure	13	2	Create CC process	
		1	Correct CC pipename	
		1	Correct robot pipename	
		1	Proper connect to CC	
		1	Proper connect to robot	
		1	Proper read of CC response	
		2	Proper read of robot response	
		1	Proper robot disconnect	
		1	Proper CC disconnect	
		2	Wait for CC.exe to quit	
Functionality	6	2	Correct initial room shown	
		2	Correct intensity shown	
		2	Handles CC errors (e.g., invalid planet)	
Printouts	6	2	CC response msg	
		2	Robot response msg	
		2	Execution time	
Misc*				

Total points: _________________

313 Homework 1 Code (Part 2)

Name: ____________________

Function	**Points**	**Break down**	**Item**	**Points**
Basic code structure	9	2	Correct U for BFS, DFS, bFS, A*	
		1	Correct D (logN overhead)	
		1	Search loop (approach #2 from slides)	
		1	Initial room in both D and U	
		3	PeekNamedPipe, dynamic buffer size	
		1	Check remainder == 0	
Functionality	14	3	BFS correct on P2	
		3	DFS correct on P2	
		3	bFS correct on P2	
		3	A* correct on P2	
		1	Handles caves without exit (900-999)	
		1	Clean termination	
Printouts	2	2	Prints exit room in hex, # of steps, distance from rover	
Misc*				

Total points: ________________

313 Homework 1 Code (Part 3)

Name: ____________________

Function	Points	Break down	Item	Points
Basic code structure	5	2	WaitForMultipleObjects(quit, semaQ)	
		1	Semaphore release	
		2	Use of mutex to protect U and D	
Speed & RAM	4	2	8000 rps with 5000 threads	
		2	Reasonable RAM utilization	
Functionality	14	3	BFS works with 5000 threads on P=6	
		3	DFS works with 5000 threads on P=6	
		3	bFS works with 5000 threads on P=6	
		3	A* works with 5000 threads on P=6	
		1	All threads quit on exit	
		1	All threads quit in caves 900-999	
Printouts	2	1	Statistics every 2 sec	
		1	Exit, distance, # of steps	
Misc*				

Extra Credit	20	10	Handle deadlock with overlapped I/O	
		10	Handle corrupted responses	

Report	25	5	Multi-core scalability analysis	
		5	Search method comparison	
		10	Distribution of distance in one planet	
		5	Average distance across planets	

Total points: ________________