EECS 281 – Winter 2014
Programming Assignment 1
Deer Hunting (Path Finding)

Hunting Season Ends February 4th, 2014, 11:55pm

Overview

In this programming assignment you will implement some basic algorithms for path-finding in C++, using them to guide a deer hunter through Michigan farms to a Big Buck.

The Hunt

Your hunt takes place on a set of farms. One farm leads to one or two adjacent farms, accessible by a ladder over the electric fence. The farms are in sequential order, and are all of the same size. The terrain of a farm and their representations consists of:

- `'.'` walkable, harvested fields
- `'#'` electric fences (not walkable!)
- `'>'` ladders up over the electric fence, leading to the next farm up the road
- `'<'` ladders down over the electric fence, leading to the previous farm down the road
- `'B'` the Big Buck
- `'S'` Grandpa's farm, the starting point

You can assume, without error checking, that there is exactly one buck, and exactly one starting point among all of the farms in the game taken together.

Your task is to plan a path for the hunter from the starting `'S'` position to the Big Buck `'B'` that does not cross any impassable `'#'` electric fences, and does not take the hunter off any of the farms.

Some farms may not be enclosed by fences; you are to treat farm edges and fences as impassable terrain.

The hunter can move north, east, south or west from any harvest field `'.'` as well as your starting location. **The hunter may not move diagonally**. Your path planning program must check that it does not move the hunter onto electric fences `'#'`.

Let the upper left corner position of a farm have coordinates (0,0). An electric fence ladder `'>'` takes the hunter to the next farm up the road (level up). If the next-farm ladder is at coordinates (*x*, *y*) on farm *k*, the hunter moves to the same coordinates (*x*, *y*) on farm *k*+1. You need to check that farm *k*+1 exists. In a similar manner, the previous farm ('<') position takes the hunter

to the previous farm down the road.

The **only** neighboring position of a ladder is the corresponding position on the farm that the ladder leads to. You need to do the same check for this neighbor position as you do other neighbor positions and take the same action based on the position type you find.

Input file format (The Farms)

The program gets its description of the farms from a file that will be read from standard input (cin). This input file is a simple text file specifying the layout of the farms. We require that you make your program compatible with two input modes: map (`M`) and coordinate list (`L`).
The reason for having two input modes is that a large percentage of the runtime of your program will be spent on reading input or writing output. Coordinate list mode exists so that we can express very large graphs with relatively few lines of a file, map input mode exists to express dense graphs (ones for which most of the tiles are not just floor space) in the smallest number of lines. Note that you should use an ostringstream, as described in discussion, to help optimize your performance when writing output - this makes a very significant runtime difference.

For both input modes ('`M`' and '`L`'):
The first line of every input file will be a single character specifying the input mode, '`M`' or '`L`'.
Note that unlike the output mode, which is given on the command-line (see below), this is part of the file.

The second line will be a single integer `N`, indicating the size of each (and every) farm of the map (each farm is `NxN` in size and all farms are the same size). Note that we do not place a limit on the magnitude of `N`; neither should your code.
The third line will be a single integer indicating the number of farms.

Comments may also be included in any input file.

Comment lines begin with `'//'` and they are allowed anywhere in the file after the first three lines. When developing your test cases, it is good practice to place a comment on line 4 describing the nature of the farms in the test case. Any farms with noteworthy characteristics for testing purposes should also be commented. By convention, a comment line identifying the farm number is placed before the map of that farm. Comments are allowed in either input mode.

Additionally - there may be extra blank/empty lines at the end of any input file - your program should ignore them. If you see a blank line in the file, you may assume that you have hit the end.

Map input mode (M):

For this input mode, the file should contain a map of each farm, in order. The farms are specified beginning with the lowest farm and working up. **The lowest farm is farm 0**; that is to say, the

farms are 0-indexed.

A valid `M` mode input example file for a map that has two 4x4 farm:

```
M
4
2
//sample M mode input file with two 4x4 farms
//farm 0
....
#...
.#..
#...
//farm 1
.B..
....
...S
#.<#
```

Coordinate list input mode (L):

The file should (after the first three lines) contain a list of coordinates for *at least* all non-walkable space coordinates in the farms. Only one coordinate should appear on a given line. We do not require that the coordinates appear in any particular order. A coordinate is specified in **precisely** the following form: `(row,col,farm,character)`. The `x` and `y` positions range from `0` to `N-1`, where `N` is the value specified at the top of the file. By default, all unspecified coordinates within the farms are of type '`.`' (floor space); however, it is not invalid to redundantly specify them to be so.

Valid coordinates (for a map with three 4x4 farms):

```
(0,1,0,#)
(2,2,2,B)
(1,3,1,.)      -- While it is valid to specify a walkable space, it is redundant!
```

Invalid coordinates (for a map with three 4x4 farms):

```
(1,2,-1,#)     -- farm -1 does not exist!
(1,2,3,.)      -- farm 3 does not exist!
(4,3,2,#)      -- Row of index 4 does not exist!
(0,1,0,F)      -- F is an invalid map character!
```

Here is a valid `L` mode input file that describes farms that are identical to those that the sample `M` input file did:

```
L
4
2
//sample L mode input file, two 4x4 farms
(1,0,0,#)
(2,1,0,#)
(3,0,0,#)
(0,1,1,B)
(2,3,1,S)
(3,0,1,#)
(3,2,1,<)
(3,3,1,#)
```

Routing schemes

You are to develop two routing schemes to get from the starting location to the Big Buck location tile:

- A queue-based routing scheme
- A stack-based routing scheme

In the routing scheme use a data structure (queue or stack) of locations within the farms. First, initialize the algorithm by adding the start position `'S'` into the data structure. Then, loop through the following steps:

1. Remove the next position from the data structure.
2. If the position you just removed is a ladder then:
    - Add the corresponding position on the farm that the ladder leads to. Up `'>'` or down `'<'` a farm in the same coordinate. **This is the only position you can move to from a ladder.**

   else:
    - Add all locations adjacent to the location you just removed that are available to move into (walkable harvested fields, ladders, or the big buck). **Add any locations that you are allowed to move to from your present location in the following order: north, east, south, and west.**
3. As you add these spaces to the data structure, check to see if any of them is the Big Buck tile `'B'`; if so, stop; else go back to step 1.

Do not add spaces that have been added to the data structure before. Remember that from a walkable harvested field tile `'.'` or your starting location 'S' you can only travel north, east, south, or west. And ladders are the only way of moving between farms.

If the data structure becomes empty before you reach the buck `'B'`, the search has failed and

there is no path to the buck.

Note: The only difference between a ladder and a walkable harvested field tile is what positions you are allowed to add into the data structure when you see it. You still have to check that you haven't visited a location before adding it.

The program must run to completion within 30 seconds of total CPU time (user + system). Note, in most cases 30 seconds is more time than you should need. See the **`time`** manpage for more information (this can be done by entering “man time” to the command line). We may test your program on very large maps (up to several million locations). Be sure you are able to navigate to the Big Buck tile in large farm maps within 30 seconds. Smaller farm maps should run MUCH faster.

Libraries and Restrictions

Unless otherwise stated, you are allowed and encouraged to use all parts of the C++ STL and the other standard header files for this project. You are **not** allowed to use other libraries (eg: boost, pthread, etc). You are **not** allowed to use the C++11 regular expressions library (it is not fully implemented on gcc) or the thread/atomics libraries (it spoils runtime measurements).

Output file format

The program will write its output to standard output (cout). Similar to input, we require that you implement two possible output formats. *Unlike input,* however, the output format will be specified through a command-line option '--output', or '-o', which will be followed by an argument of `M` or `L` (`M` for map output and `L` for coordinate list output). See the section on command line arguments below for more details.

For both output formats, you will show the path you took from start to finish. In both cases you should first print the number of rows/cols in each farms and then the number of farms of the map.

Map output mode (M):

For this output mode, you should print the map of the farms, replacing existing characters as needed to show the path you took. Beginning at the starting location, overwrite the characters in your path with `'n'`, `'e'`, `'s'`, `'w'`, `'d'` (farm down the road), and `'u'` (farm up the road) to indicate which tile you moved to next. Do not overwrite the Big Buck `'B'` at the end of the path, but do overwrite the `'S'` at the beginning. For all spaces that are not a part of the path, simply reprint the original farm map space. *You should discard all existing comments from the input file for the output file*. However, do create comments to indicate the farm numbers and format them as shown below:

Thus, for the sample input file specified earlier, using the queue-based routing scheme and map (`M`) style output, you should produce the following output:

```
4
2
//farm 0
....
#...
.#..
#...
//farm 1
.Bww
...n
...n
#.<#
```

Using the same input file but with the stack-based routing scheme, you should produce the following output:

```
4
2
//farm 0
....
#...
.#..
#...
//farm 1
eB..
n...
nwww
#.<#
```

We have highlighted the modifications to the output in red to call attention to them; do not attempt to color your output (this isn't possible, as your output must be a plain text file).

Coordinate list output mode (L):

For this output mode, you should print only the coordinates that make up the path you traveled. You should print them in order, from (and including) the starting position to the '`B`' (but you should not print the coordinate for '`B`'). The coordinates should be printed in the same format as they are specified in coordinate list input mode `(row,col,farm,character)`. The character of the coordinate should be `'n'`, `'e'`, `'s'`, `'w'`, `'d'` or `'u'` to indicate spatially which tile is moved to next. You should discard all comments from the input file, but you should add one comment on

line 3, just before you list the coordinates of the path that says "`//path taken`".

The following are examples of correct output format in (`L`) coordinate list mode that reflect the same solution as the Map output format above:

For the queue solution:

```
4
2
//path taken
(2,3,1,n)
(1,3,1,n)
(0,3,1,w)
(0,2,1,w)
```

For the stack solution:

```
4
2
//path taken
(2,3,1,w)
(2,2,1,w)
(2,1,1,w)
(2,0,1,n)
(1,0,1,n)
(0,0,1,e)
```

There is only one acceptable solution per routing scheme for each map of the farms. If no valid route exists, the program should simply reprint the farms with no route shown for Map output mode, and should have no coordinates listed after the "`//path taken`" comment in coordinate List output mode.

Please note that the mode of input and output can vary. That is, the input mode may be Coordinate mode, but the output may be requested in Map mode and vise-versa. They may also be, but are not guaranteed to be, the same.

Command line arguments

Your program should take the following case-sensitive command line options (when we say a switch is "set", it means that it appears on the command line when you call the program):

- **--stack, -s**: If this switch is set, use the stack-based routing scheme.
- **--queue, -q:** If this switch is set, use the queue-based routing scheme.
- **--output (M|L), -o (M|L):** Indicates the output file format by following the flag with an `M`

(map format) or `L` (coordinate list format). If the --output option is not specified, default to map output format (M), if --output is specified on the command line, the argument (either `M` or `L`) to it is required. See the examples below regarding use.

- **--help, -h:** If this switch is set, the program should print a brief help message which describes what the program does and what each of the flags are. The program should then exit(0) or return 0 from main.
- Note: When we say --stack, or -s, we mean that calling the program with --stack does the same thing as calling the program with -s. See **getopt** for how to do this.

Legal command line arguments must include exactly one of --stack or --queue (or their respective shortforms -s or -q). If none are specified or more than one is specified, the program should print an informative message to standard error (cerr) and exit(1).

Examples of legal command lines:

- ./proj1 --stack < infile > outfile
  - This will run the program using the stack algorithm and map output mode.
- ./proj1 --queue --output M < infile > outfile
  - This will run the program using the queue algorithm and map output mode.
- ./proj1 --stack --output L < infile > outfile
  - This will run the program using the stack algorithm and coordinate list output mode.

Note that as discussed in discussion, we are using input and output redirection here. While we are reading our input from a file and sending out output to another file in this case, we are <u>NOT</u> using file streams! The operating system makes calls to cin read the input file and it makes calls to cout append to the output file. Come to office hours if this is confusing!

Examples of illegal command lines:

- ./proj1 --queue -s < infile > outfile
  - Contradictory choice of routing
- ./proj1 < infile > outfile
  - You must specify either stack or queue

Test cases

It is extremely frustrating to turn in code that you are 'certain' is functional and then receive half credit. We will be grading for correctness primarily by running your program on a number of test cases. If you have a single silly bug that causes most of the test cases to fail, you will get a very low score on that part of the project *even though you completed 95% of the work.* Most of your grade will come from correctness testing. Therefore, it is imperative that you test your code thoroughly. To help you do this we will require that you write and submit a suite of test cases that thoroughly test your project.

Your test cases will be used to test a suite of buggy solutions to the project. Part of your grade will be based on how many of the bugs are exposed by your test cases. (We say a bug is *exposed* by a test case if the test case causes the buggy solution to produce different output from a correct solution.)

Each test case should be an input file that describes a map of farms in either map (`M`) or coordinate list (`L`) format. Each test case file should be named *test-n.txt* where 0 < n <= 15 for each test case. All test cases will be run in both queue mode and stack mode. Test cases may have no more than 10 farms, and the size of a farm may not exceed 8x8. You may submit up to 15 test cases (though it is possible to get full credit with far fewer test cases). Note that the tests the autograder runs on your solution are **NOT** limited to 10x8x8; your solution should not impose any size limits (as long as sufficient system memory is available).

Errors you must check for

A small portion of your grade will be based on error checking. You must check for the following errors:

- Input errors: illegal map characters.
- For coordinate list input mode, you must check that the row, column, and farm numbers of each coordinate are all valid positions.
- More or less than one --stack or --queue on the command line. You may assume the command line will otherwise be correct (this also means that we will not give you characters other than '`M`' or '`L`' to --output).

In all of these cases, print an informative error message to standard error and exit(1).
You do not need to check for any other errors.

Assumptions you may make

- You may assume we will not put extra characters after the end of a line of the map or after a coordinate.
- You may assume that coordinates in coordinate list input mode will be in the format `(row,col,farm,character)`.
- You may assume that there will be exactly one start location 'S' and exactly one Big Buck tile 'B' in the map.
- You may assume that we will not give you the same coordinate twice for the coordinate list input mode.
- You may assume the input mode line and the integer dimensions of the farms on lines two and three at the beginning of the input file will be by themselves, without interspersed comments, and that they will be correct.

Submission to the Autograder

Do all of your work (with all needed files, as well as test cases) in some directory other than your home directory. This will be your "submit directory". Before you turn in your code, be sure that:

- You have deleted all .o files and your executable(s). Typing '`make clean`' shall accomplish this.
- Your makefile is called Makefile. Typing '`make -R -r`' builds your code without errors and generates an executable file called `proj1`. (Note that the command line options -R and -r disable automatic build rules, which will not work on the autograder).
- Your Makefile specifies that you are compiling with the gcc optimization option -O3. This is extremely important for getting all of the performance points, as -O3 can often speed up code by an order of magnitude. You should also ensure that you are not submitting a Makefile to the autograder that compiles with the debug flag, -g, as this will slow your code down considerably. Note: If your code "works" when you don't compile with -O3 and breaks when you do, it means you have a bug in your code!
- Your test case files are named test-n.txt and no other project file names begin with test. Up to 15 pairs of tests may be submitted.
- The total size of your program and test cases does not exceed 2MB.
- You don't have any unnecessary files or other junk in your submit directory and your submit directory has no subdirectories.
- Your code compiles and runs correctly using version 4.7.0 of the g++ compiler. This is available on the CAEN Linux systems (that you can access via login.engin.umich.edu). Even if everything seems to work on another operating system or with different versions of GCC, the course staff will not support anything other than GCC 4.7.0 running on Linus. Note: In order to compile with g++ version 4.7.0 on CAEN you must put the following at the top of your Makefile:

```
PATH := /usr/um/gcc-4.7.0/bin:$(PATH)
LD_LIBRARY_PATH := /usr/um/gcc-4.7.0/lib64
LD_RUN_PATH := /usr/um/gcc-4.7.0/lib64
```

Turn in all of the following files:

- All your .h and .cc or .cpp files for the project
- Your Makefile
- Your test case files

You must prepare a compressed tar archive (.tar.gz file) of all of your files to submit to the autograder. One way to do this is to have all of your files for submission (and nothing else) in one directory. Go into this directory and run this command:
dos2unix -U *; tar czf ./submit.tar.gz *.cpp *.h *.cc *.c Makefile test*.txt
This will prepare a suitable file in your working directory.

Submit your project files directly to either of the two autograders at: https://g281-1.eecs.umich.edu/ or https://g281-2.eecs.umich.edu/. You can safely ignore and override any warnings about an invalid security certificate. **Note that when the autograders are turned on and accepting submissions, there will be an announcement.** The autograders are identical and your daily submission limit will be shared (and kept track of) between them. You may submit up to three times per calendar day with autograder feedback. For this purpose, days begin and end at midnight (Ann Arbor local time). **We will count only your last submission for your grade**. We realize that it is possible for you to score higher with

earlier submissions to the autograder; however this will have no bearing on your grade. We strongly recommend that you use some form of revision control (ie: SVN, GIT, etc) and that you 'commit' your files every time you upload to the autograder so that you can always retrieve an older version of the code as needed. Please refer to your discussion slides and CTools regarding the use of version control.

Please make sure that you read all messages shown at the top section of your autograder results! These messages will help explain some of the issues you are having (such as losing points for having a bad Makefile).

Grading

80 points -- Your grade will be primarily based on the correctness of your algorithms. Your program must have correct and working stack and queue algorithms and support both types of input and output modes. **Additionally:** Part of your grade will be derived from the runtime performance of your algorithms. Fast running algorithms will receive all possible performance points. Slower running algorithms may receive only a portion of the performance points. A leader board will be posted on the course website, and will be updated frequently during the project. You may track your progress relative to other students and instructors there.
20 points -- Test case coverage (effectiveness at exposing buggy solutions).

Grading will be done by the autograder.

We also reserve the right to deduct up to 5 points for bad programming style.

Coding style

A small portion of your grade may be derived from having good coding style. Among other things, good coding style consists of the following:

- Clean organization and consistency throughout your overall program
- Proper partitioning of code into header and cpp files
- Descriptive variable Names and proper use of C++ idioms
- Effective use of library (STL) code
- Omitting globals, unnecessary literals, or unused libraries
- Effective use of comments
- Reasonable formatting - e.g an 80 column display
- Code reuse/no excessive copy-pasted code blocks

Effective use of comments includes stating preconditions, invariants, and postconditions, explaining non-obvious code, and stating big-Oh complexity where appropriate.

It is **extremely helpful** to compile your code with the gcc options: -Wall -Wextra -pedantic. This will help you catch bugs in your code early by having the compiler point out when you write code

that is either of poor style or might result in behavior that you you did not intend.

Empirical efficiency

We will check for empirical efficiency both by measuring the memory usage and running time of your code and by reading the code. We will focus on whether you use unnecessary temporary variables, whether you copy data when a simple reference to it will do, whether you use an O(n) algorithm or an $O(n^2)$ algorithm.

Hints and advice

- Design your data structures and work through algorithms on paper first. Draw pictures. Consider different possibilities *before* you start coding. If you're having problems at the design stage, come to office hours. After you have done some design and have a general understanding of the assignment, re-read this document. Consult it often during your assignment's development to ensure that all of your code is in compliance with the specification.
- Always think through your data structures and algorithms before you code them. It is important that you use efficient algorithms in this project and in this course, and coding before thinking often results in inefficient algorithms.
    - If you are considering linked lists, be sure to review the lecture slides or measure their performance against vector first (theoretical complexities and actual runtime can tell different stories).
- Only print the specified output to standard output.
- You may print whatever any diagnostic information you wish to standard error. However, make sure it does not scale with the size of input, or your program may not complete within the time limit for large test cases.
- If the program does find a route, be sure to have main return 0 (or call exit(0)). If the input is valid but no route exists, also have main return 0.
- *This is not an easy project.* ***Start it immediately!***

Have fun coding!