mp_mazes

Maddening Mazes

Extra credit: Apr 08, 23:59 PM
Due: Apr 15, 23:59 PM

Doxygen

Partner MP mp_mazes is a **partner MP**!

- Part 1 and Part 2 of this MP can be completed with a partner!
- The creative "Part 3" must be completed by yourself and must be unique (and different from your partner's work).

You should denote who you work with in the PARTNERS.txt file in mp_mazes. If you worked alone, include only your NetID in PARTNERS.txt.

Goals and Overview

In this MP you will:

- Implement the disjoint sets data structure.
- Create a program to generate random mazes.
- Applying a DFS traversal to a maze structure
- Represent a maze and its solution on a `PNG`.

Checking Out the Code

From your CS 225 git directory, run the following on EWS:

```
git fetch release
git merge release/mp_mazes -m "Merging initial mp_mazes files"
```

If you're on your own machine, you may need to run:

```
git fetch release
git merge --allow-unrelated-histories release/mp_mazes -m "Merging initial mp_mazes files"
```

The `mp_mazes` directory will contain sample output including all 4 possible 2x2 mazes and one 50x50 maze. We encourage you to test your code by writing your own `main.cpp` that uses your classes in different ways.

Assignment Requirements

These are strict requirements that apply to **both** parts of the MP. Failure to follow these requirements may result in a failing grade on the MP.

- You are required to comment the MP as per the commenting standard described by the Coding Style Policy.
- You must name all files, public functions, public member variables (if any exist), and executables **exactly** as we specify in this document.
- Your code must produce the **exact** output that we specify: nothing more, nothing less. Output includes

standard and error output and files such as Images.
- Your code must compile on the EWS machines using `clang++`. Being able to compile on a different machine is **not** sufficient.
- Your code must be submitted correctly by the **due date and time**. Late work is not accepted.
- Your code must not have any memory errors or leaks for full credit. ASAN tests will be performed separately from the functionality tests.
- Your public function signatures must match ours **exactly** for full credit. If using different signatures prevents compilation, you will receive a zero. Tests for `const`-correctness may be performed separately from the other tests (if applicable).

Assignment Description

You will be implementing a `Disjoint` set data structure and then implementing a random maze generator and solver. The assignment is broken up into the two following parts:

- Part 1 — The `DisjointSets` data structure and the beginning of a story for use later in the semester.
- Part 2 — The `SquareMaze` random maze generator and solver.

As usual, we recommend implementing, compiling, and testing the functions in Part 1 before starting Part 2. Submission information is provided for each part in the respective sections below.

Part 1a: The `DisjointSets` data structure

The `DisjointSets` class should be declared and defined in `dsets.h` and `dsets.cpp`, respectively. Each `DisjointSets` object will represent a family of disjoint sets, where each element has an integer index. It should be implemented with the optimizations discussed in lecture, as up-trees stored in a single `vector` of `int`s. Specifically, use path compression and union-by-size. Each element of the vector represents a node. (Note that this means that the elements in our universe are indexed starting at 0.) A nonnegative number is the index of the parent of the current node; a negative number in a root node is the negative of the set size.

Note that the default compiler-supplied Big Three will work flawlessly because the only member data is a `vector<int>` and this vector should initially be empty.

The `addelements` function

See the Doxygen for this function.

The `find` function

See the Doxygen for this function.

The `setunion` function

See the Doxygen for this function.

The `size` function

See the Doxygen for this function.

Testing Part 1a

The following command can be used to compile the `DisjointSets` test executable:

```
make testdsets
```

The following command can be used to run the test executable:

```
./testdsets
```

Provided Catch test cases are available as well by running:

```
make test
./test
```

Part 1b: Create a Choose Your Own Adventure

> **Solo Portion** This creative part of the MP must be completed individually and must be significantly different from your partner's creative work.

An interactive story presents a reader with a narrative and allows her a choice of options to advance the story. For example, the story may read:

Zoey walks into a long hallway, where she sees three doors. The first, a large old oak door, is slightly ajar; the second, a narrow passageway that looks as though it leads into another hallway; and the third, a sterile metal door that is propped open with a large stone. Uncertain which way to travel, she takes a second to consider each choice.

From the above narrative, the reader chooses where Zoey advances next through an interactive choice:

- *After a moment, Zoey chooses to travel through the oak door.*
- *With a bit of fear, Zoey chooses to travel through the narrow passageway.*
- *Walking to the end of the hallway, Zoey chooses to travel through sterile metal door.*

At the core, every interactive story makes a graph. Each **vertex** is a narrative and each **edge** is a choice that leads from one narrative to another. In this activity, you will write your own interactive story and construct a graph of your story.

You do not need to do anything with the graph or build any graph algorithms! You will be learning about graphs in CS 225 and this story will be part of a course-wide story-telling project.

Story Format Specification

Your story must consist of several Markdown (`.md`) files that has one narrative and any options to move forward in the story. For example, `waf-zoey.md` is already in your `story_data` folder:

```
Zoey walks into a long hallway, where she sees three doors. The first, a large old oak
door, is slightly ajar; the second, a narrow passageway that looks as though it leads into
another hallway; and the third, a sterile metal door that is propped open with a large
stone. Uncertain which way to travel, she takes a second to consider each choice.

# waf-zoey-oak
After a moment, Zoey chooses to travel through the oak door.

# waf-zoey-narrow
With a bit of fear, Zoey chooses to travel through the narrow passageway.

# waf-zoey-metal
Walking to the end of the hallway, Zoey chooses to travel through sterile metal door.
```

Note the following bits:

- The first section of the file is the narrative.

- Each interactive choice is defined by a line starting with `#`, followed by the key (filename, without `.md`) for the next vertex.
- The text for each interactive choice follows the `#` header line.

Story Requirement

To complete Part 1b, your story must feature **at least one** of these following 12 people:

- **Bill Gates**, founder of Microsoft, author, and philanthropist - Wikipedia: Bill Gates
- **Taylor Swift**, Taylor Swift (and actually really actually made the list!!) - Wikipedia: Taylor Swift
- **Lebron James**, basketball superstar - Wikipedia: Lebron James
- **Beyoncé**, singer, songwriter, and actress - Wikipedia: Beyoncé
- **Albert Einstein**, theoretical physicist, theory of relativity - Wikipedia: Albert Einstein
- **Sheryl Sandberg**, chief operating officer (COO) of Facebook, author - Wikipedia: Sheryl Sandberg
- **Karlie Kloss**, "Kode with Klossy", model, and entrepreneur - Wikipedia: Karlie Kloss
- **Elon Musk**, Tesla, SpaceX, and more - Wikipedia: Elon Musk
- **Grace Hopper**, pioneer in programming, invented early compiler related tools - Grace Hopper
- **Steve Jobs**, former CEO of Apple Inc. - Wikipedia: Steve Jobs
- **Emma Watson**, actress, activist - Wikipedia: Emma Watson
- **Lil Yachty**, rapper, singer, songwriter and performer at the UIUC Spring Jam - Wikipedia: Lil Yachty

You do not need to **only** feature these people (*you can feature others!*) but your story must feature **at least one** of these people. This will help you find others who have similar stories to join with in the next MP.

In addition to the featured person(s), the following conditions must be met:

- You must delete `waf-*.md` files – write your own story! :)
- Your interactive story must have at least 8 narratives (8 files, 8 vertices). You are encouraged to have more!
- Your interactive story must have at least 13 choices (13 edges). You are encouraged to have more!
- At least one narrative must have at least 3 choices (one vertex must have at least 3 outbound edges).
- No narrative can have more than 10 choices (no vertex may have more than 10 outbound edges).
- Exactly one narrative must be the beginning of the story (one vertex must have **no incoming edges**, you can never return all the way to the start).
- At least one narrative must be the ending of the story (an ending vertex will have no outbound edges). You can have multiple endings.
- Your story must tell a story. It cannot simply be letters/words that lead to other letters/words. Be creative, have fun.
- **Your story must be something you would be proud to show your parents and have your name attached to it. If your story is beyond "G/PG rated", you will get a 0 on the full assignment.**

Testing - Part 1b

For grading purposes, your story files must be committed in the folder `story_data`. Because these are new files, you will need to manually git add them (`git add -u` will not work). You can add all changes to your story files by using:

```
git add story_data
```

This needs to be run any time you add a new story file. Otherwise, you can simply run `git add -u` as normal.

We have provided a Python script `story_validator.py` for helping you to verify that your story matches the requirements above. It requires Python3.6 or above. On EWS, you will need to run the following command to load a new enough version of Python:

```
module load python3/3.7.0
```

To run the script, simply pass it the path to your story folder. For example, if you are running the script in your `mp_mazes` folder, you would run:

```
python3 story_validator.py story_data
```

Grading Information — Part 1

The following files are used to grade mp_mazes:

- `dsets.cpp`
- `dsets.h`
- All `.md` files in `story_data/`

All other files including your testing files will not be used for grading.

Part 2: The `SquareMaze` random maze generator and solver

The `SquareMaze` class should be declared and defined in `maze.h` and `maze.cpp`, respectively. Each `SquareMaze` object will represent a randomly-generated square maze and its solution. **Note that by "square maze" we mean a maze in which each cell is a square; the maze itself need not be a square.** As always, we recommend reading the whole specification before starting.

> **`setWall` and `canTravel`** You should triple check that `setWall` and `canTravel` function exactly according to spec, as an error in these functions will not be caught by making your own mazes but can cost you a majority of the points during grading.

Videos

- Cycle Prevention / Detection

The `makeMaze` function

See the Doxygen.

The `canTravel` function

See the Doxygen.

The `setWall` function

See the Doxygen.

The `solveMaze` function

See the Doxygen.

The `drawMaze` function

See the Doxygen.

The `drawMazeWithSolution` function

See the Doxygen.

Testing Part 2

Square Maze Testing

Since your mazes will be randomly generated, we cannot provide you with any sample images to diff against. However, we have provided you with all four possible 2x2 mazes. If you have your program create and solve a 2x2 maze, the resulting image (with solution) should match one (and only one) of the provided images `m0.png`, `m1.png`, `m2.png`, and `m3.png`. We strongly suggest that you diff against these to make sure that you have formatted the output image correctly.

We provide some basic code to test the functionality of `SquareMaze`.

The following command can be used to compile the `SquareMaze` test executable:

```
make testsquaremaze
```

The following command can be used to run the test executable:

```
./testsquaremaze
```

You can compare the console output of your program with the expected by comparing it with the file `soln_testsquaremaze.out`.

Part 3: A Final Creative Component

> **Solo Portion** This creative part of the MP must be completed individually and must be significantly different from your partner's creative work.

Similar to Part 2 of this MP, we are leaving the design of the creative component completely up to you!

Use `main.cpp` to generate a file, `creative.png`, that contains a creative maze. The only requirement is that this maze **must not** be a rectangular maze similar to Part 2. For example, what if your maze was a circle? Or in the shape of a block I? What if you maze made us of ideas from MP1, MP2, MP4, or MP5? Be creative. Have fun.

As you complete Part 3, ensure that your Part 2 still continues to function as described. This means you may need to add a `drawCreativeMaze()` function instead of modifying your `drawMazeWithSolution()` or other function. You should find many of your existing helper functions still useful.

> **You just made something awesome that never existed before -- you should share your art (but do not have to)!**
> *If you share your art on Facebook, Twitter, or Instagram with **#cs225**, I will 👍 or ❤ the post as soon as I see it. I think many of your peers will too!* — Wade

Catch Test Suite

Additional unit tests similar to how we will grade your MP are provided for you in `tests/test_part2.cpp`. Once you have completed Part 2, you may **uncomment** the Part 2 tests and re-compile to include Part 2 tests:

```
// uncomment test_part2.cpp
make test
./test
```

Runtime Concerns

You should strive for the best possible implementation. This MP can be implemented so that the given `testsquaremaze.cpp` runs in less than a quarter of a second on the EWS linux machines. To have a high probability of finishing within the time constraints of the grading script, make sure you can run the given `testsquaremaze.cpp` in under 3 seconds on an unencumbered machine. You can time mp_mazes by running the command `time ./testsquaremaze`.

Grading Information — Part 2

- We will use `canTravel` to reconstruct your randomly generated maze in our own maze class, to check that it's a tree and to compare your implementation to a correct implementation of this MP. This will require `height*width*2` calls to `canTravel`. Therefore, it is very important that `canTravel` works, and works quickly (constant time). If it doesn't work, you will lose a lot of points.
- We will use `setWall` to replace your maze with our own, for the purpose of testing all of your other functions independently of your `createMaze`.

> **`setWall` and `canTravel`** You should triple check that `setWall` and `canTravel` function exactly according to spec, as an error in these functions will not be caught by making your own mazes but can cost you a majority of the points during grading.

The following files are used to grade mp_mazes:

- `dsets.cpp`
- `dsets.h`
- `maze.cpp`
- `maze.h`
- `main.cpp`, for Part 3 only

All other files will not be used for grading.