Maze w/ Multithreading

Joey Domino

Well well well. Look what we have here. I have another paper to write. In this document, I will cover the inner workings of my multithreaded maze system. This includes how the system begins, how threads are created and communicated between, how data moves through the system, and the challenges therein. I will also be covering how I would improve my system as the maze search isn't perfect.

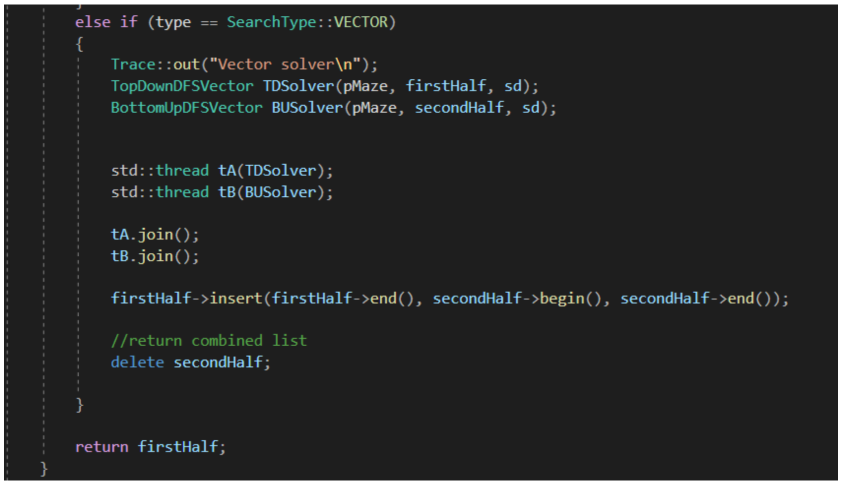
Current system

Let's start off with an overview. When the MTMazeStudentSolver is called, it first checks if it is using an array or vector implementation. Either option runs a similar system, but with a different storage type. The array storage is faster, but has size limitations while the vector is slower, but doesn't have an issue with storage space.

Once the choice is made, two threads are created: TopDownDFS and BottomUpDFS. These threads do their work, which I will expand upon below, and are joined back to the MTMazeStudentSolver to await their completion. Each thread populates a portion of the solved maze via a list of directions (either an array or vector). The directions are then combined into a full list after each thread is joined back to MTMSS and returned to the main script as the solution. This is a bird's eye view of the process. Let's dive deeper now to understand the inner workings of the threads, how they communicate, and how data traverses the threads.

Thread Creation

MTMazeStudentSolver creates the threads by calling std::thread x(topDown) and std::thread x(bottomUp). This kicks off each thread to run the given functors created within each solver. The threads then do their jobs. One thread starts from the top of the maze while the other begins at the end. Each then traverse the maze in the same manner, but from opposing sides. This should, in theory, cut the traversal time in half. MTM immediately calls join() on both threads, forcing the method to wait for the threads to finish before fuses the solution lists.



```
    else if (type == SearchType::VECTOR)
    {
        Trace::out("Vector solver\n");
        TopDownDFSVector TDSolver(pMaze, firstHalf, sd);
        BottomUpDFSVector BUSolver(pMaze, secondHalf, sd);


        std::thread tA(TDSolver);
        std::thread tB(BUSolver);

        tA.join();
        tB.join();

        firstHalf->insert(firstHalf->end(), secondHalf->begin(), secondHalf->end());

        //return combined list
        delete secondHalf;

    }

    return firstHalf;
}
```

Communication between threads

Thread communication is pretty simple in this implementation. The threads traverse the maze on their own, just from opposing starting positions. The communication comes when one thread touches the other or reaches the end/start of the maze (if the thread starts at the end/beginning respectively). This flips an atomic boolean flag within a SharedData class to tell both threads to end processing. The end position for whichever thread hits the other first is saved as well. In this case we'll call the first finishing thread, thread A.

The end position is used to roll back thread B's movements to line up with where thread A stopped when touching thread B's path. This ensures that the returned lists are accurate and have no duplicate positions.

Data movement

I have covered data movement already, but I'll really quickly go through it again here. Main calls MTMazeStudentSolver with the maze to be solved and an enumeration specifying which data type to use for storage, array or vector. MTMMazeStudentSolver then makes two threads to traverse the maze from the start and end positions. Each thread builds their path via a vector of position nodes. The position nodes hold the direction moved (North, South, East, or West). Once one thread hits the path of the other, a shared flag is flipped to tell both mazes to finish, clean up, and store the end position of the thread that touched the other.

The cleanup process includes retracing movements in thread B until it hits the end position of thread A or vice versa. The lists are then blended together within MTMazeStudentSolver and returned to maze to check the results.

Challenges

I thought the maze traversal assignment would be way easier than the jetson's playback system, but boy was I wrong. Having such an open ended project made it a bit daunting. I found it hard to figure out a plan of attack with all my other responsibilities going on. Time management was definitely the worst aspect of building this out. However, I'll focus on the main challenges of actually building this out.

First off, parsing the data was a large task. Stepping through the list wasn't too bad, but I definitely hit the edge cases discussed in class where one thread may not be where the other is on contact. I still have a problem where larger mazes are missing a small number of nodes. I'm not sure why that's the case and I'm out of time to look into it.

I hit a surprising wall with the storage of the positions. Using a vector to store all the locations seemed to work (until I hit larger mazes), but it was slower. So I attempted to use an array instead to store the positions. This was faster, but I hit a different wall when building to scale. I would error out when my arrays were at a size of roughly 60,000 positions. I kept in the array option due to the speed, but made the vector option default as it was more reliable.

I also incorrectly setup my threads. As I stated in the thread creation portion, I forced the threads to join immediately after creating them. This essentially made the threads moot as I forced MTMazeStudentSolver to wait for thread A to finish and then thread B. The threads weren't properly working in tandem.

There's a LOT I would do different if I had more time, but that's not the case. I have listed below all of the things I would change about my current implementation. Hopefully this coverage will show my understanding of the procedures involved with multithreading. There is not enough hours in the day (OR NIGHT. SO MANY ALL NIGHTERS!!) ED!

EDIT:

I took out the Positions vector list to instead use the pChoices vector list already in use. This solved my storage issue and speed issue. This could be related to the parallelization problem we covered. Saving in auxiliary storage may have helped speed things up.

Maze not working? What happened?

I actually fixed up the maze, so the below is not as important, but I still would have liked to implement a lot of this if I had more time.

Why am I in this situation?

First: a bit of hard truth. I bit off WAY more than I could chew this quarter. I started a new job in June that takes 8-10 hours of my day. I was also in RTSDII which ended up taking the majority of my free time. I ended up missing classes because I was working on the homework during the lectures to try and keep up. It made me fall behind in both. That's not an excuse, but just the fact of the matter. It's why you didn't hear much from me in each lecture. I was fervently working. I kept hitting walls.

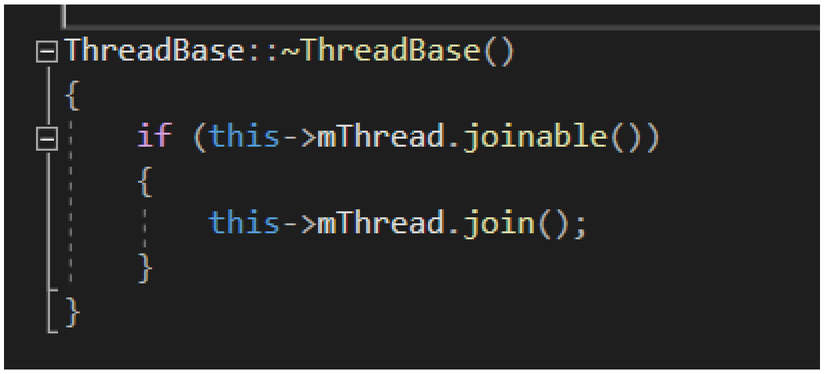
As for the maze; it's two pronged. I thought the maze would be a lot easier to handle than the jetson's assignment. I thought I could get away with a simpler implementation and just not win any speed tests. Boy was I wrong. I built the maze system to be a simple 2 threaded process to parse through the maze from the start and ending positions and meet somewhere in the middle.

The main problem, I believe, is that I joined my threads in MTMazeStudentSolver when I should have had the join in the destructor of the class holding each process. The join being incorrectly placed really slowed down the system as it essentially made thread B wait for thread A to finish to process. It ruined the whole thing! I should have used what I did in the Jetsons program, but I thought I could keep it simple. Ya fool.

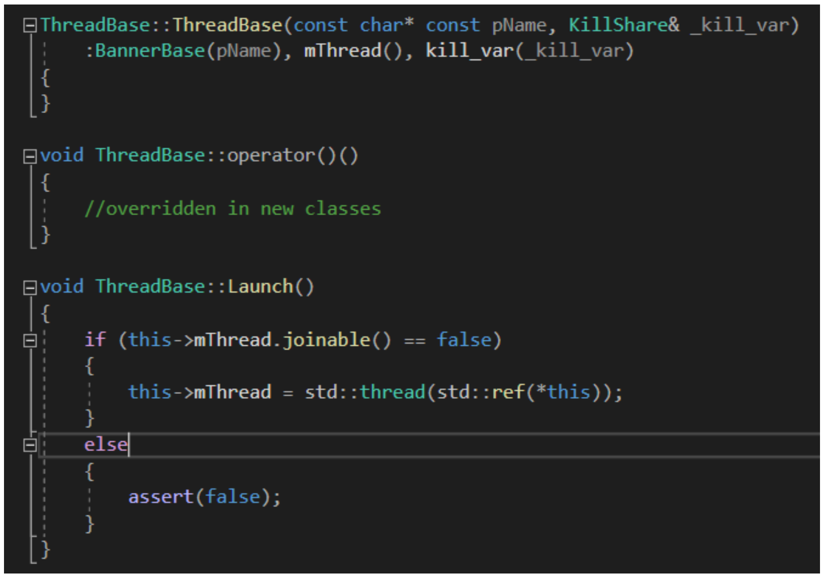
So the basic problem is how I closed my two threads and lack of time. The 2nd one isn't able to be changed, so how would I change the process of using my threads to solve the maze?

Diagram every problem and how I'd solve it

As I briefly mentioned above, I would move where my joins are called to the thread classes destructors. To clean this up even more, I would make a ThreadBase class that houses all threads I make. This way, every created thread gets a new thread on creation automatically and the thread is rejoined on destruction. That would allow my threads to run on their own until told otherwise whether by a bool or status update. I have taken some shots of code form the jetson's system to show what I mean.



```
ThreadBase::~ThreadBase()
{
    if (this->mThread.joinable())
    {
        this->mThread.join();
    }
}
```



```
ThreadBase::ThreadBase(const char* const pName, KillShare& _kill_var)
    :BannerBase(pName), mThread(), kill_var(_kill_var)
{
}

void ThreadBase::operator()()
{
    //overridden in new classes
}

void ThreadBase::Launch()
{
    if (this->mThread.joinable() == false)
    {
        this->mThread = std::thread(std::ref(*this));
    }
    else
    {
        assert(false);
    }
}
```

To help speed up the application, I would create a new thread system to create threads at decision nodes to hunt down dead ends from each decision. This would mean for every move made, there would be three threads running in tandem searching every possible direction we can go for a given node. This could lead to an issue with too many threads running at once if every decision node creates a new thread. So my current thinking is to make something like the waveBuffThread from Jetsons. WaveBuffThread has a limit of 20 threads to play the incoming data. I would do something similar for parsing dead ends. Keep making threads until we fill up to the cap, then wait till dead ends are found to free up more space.

```
//Can we move more than one direction?
If(currPosition.moveDecisions > 1)
{
        //Some hard limit. Let's say 20 threads is cap
        If(DeadEndThread::threadCount != DEAD_END_CAP)
        {
                //We can make one!
                DeadEndThread* pDEInstance = new DeadEndThread(currPosition);
                //Track the count of DE threads somehow
                pDEThreadsCount++;

                //start thread
                pDEInstance.Launch();
        }
}
```

The DeadEndThreads will have ThreadBase as their parent like all other threads would with these changes, so they will join on destruction cleanly. The dead end threads will work much the same as the normal process of stepping through nodes and moving through decisions, making new threads as they keep branching up until the hard cap, then they traverse just like the main loop. The idea here is to split every possible decision to a new thread and only return the data of the path that isn't a dead end to the main process so it knows which way to go.

My largest problem was storage. For whatever reason my nodes weren't being saved properly and I really have no clue why. I'm not sure why the vectors aren't cooperating, but I believe If I switched to the arrays and focused on them, I could get a pooling system working to store larger amounts of position data.

I think I would add positional data within the shared data class and update pools of positions with topDown and bottomUp's positional data as they move, ignoring the dead ends.

```
SharedData
{
Position top1positions[30000];
Position top2positions[30000];
Position bottom1positions[30000];
Position bottom2positions[30000];
}
```

```
If(this->positions == filled)
{
        If(sharedData.top1Positions == empty)
        {
        sharedData.top1Positions = this->positions
        this->positions.empty();
        }
        Else if(sharedData.top2Positions == empty
        {
        sharedData.top2Positions = this->positions;
        this->positions.empty();
        }
}

//Then do the same for bottom.
//Merge all 4 position data at the end into the solution path.
//Same way as it's currently done, just 2 more fields.
```

Update: This isn't necessary at all. I'll keep it here, but I really think it won't really achieve much of anything. Using an array was a mistake.

It's a crude setup, but may get around the problem I was having. Honestly multithreading wasn't the issue with this assignment aside from the lazy thread joining I did in MTMazeStudentSolver.

Edge cases screwed me up and I ran out of time while working on other projects and work. I know I could get this working with more time, but I'm all out. It is what it is. Hopefully I explained things relatively well enough to show I know what I'm talking about to some degree.

I really enjoyed this class, but man this was NOT a small amount of coding!!! This was so much! I agree that there was a lot more understanding principles than actual coding, but this was still a pretty rough class. I could have MAYBE done both this and RTSDII together if I didn't have a full time job. I just hope I passed both classes with a B. I'll be happy to move on with that. Fingers crossed and see you next quarter.

EDIT:

I fixed it! Removed the Positions vector list for both bottomUp and topDown paths to instead use the existing pChoices directions list solved a lot of problems. I also took the DFS process you used in STMazeSolverDFS to fix up my StudentSolFoundSkip catch also helped. Lordy today was a nightmare. I can't believe I did all this. I need a break. See you in a few weeks!