CSC 412 - A4 Report

Ryan Brooks

This assignment has definitely been my favorite so far; it touches upon so many areas of Computer Science from multiple data structures, a pathfinding algorithm, a bit of dynamic programming, and concurrent CPU processes.

When starting out, I knew the assignment would grow in scale and complexity through each version, so I would have to make careful decisions about the setup of my project. Throughout this report, I will discuss my process for implementing the assignment, the data structures, algorithms, and abstraction choices made, difficulties encountered, details of forking the children and grandchildren processes, and the limitations of my program.

Please see the README file for sources, file organization, and extra credit completed.

Implementation Details & Difficulties

I wanted to separate creating the graph and cost matrix away from any pathfinding, so I decided on a Graph class and a pathfinder.cpp file to handle various steps of the process. I'd also need a testing module, which I'll discuss later. Main.cpp would walk through each step of the process, relying on functions defined in pathfinder.cpp.

I first had to construct the cost grid, which was simple. I needed to do error checking, but due to the nature of taking each step of the program one piece at a time, I could rely on invariants of previous steps to do most of the work for me. Once I had the cost grid, I needed to create the graph. At this point I only had nodes, so in the constructor I did some error checking and created a vector of Node structs, an Enum to handle storing a node's index and position (as a pair). This struct would be used only in the Graph class but in other parts I could just refer to nodes by their index (based on how they were read in from the text file).

Once I had the nodes, this is where the real error checking began. I could check to see if the given graph could be overlaid on the cost grid–meaning the graph "fit" without any overflow. Knowing that I had a compatible graph and cost grid, I needed to actually construct the graph's edges. I chose to use an adjacency list to store the connected 3 nodes it found with the Manhattan distance because I felt like an adjacency matrix would be too space inefficient for large graphs, and even more so if they were sparse.

Now that my nodes were connected, I began to find the valid paths on the graph–or a path that includes 3 to 5 nodes and goes from the provided starting node to ending node. I used a recursive DFS approach. Due to this meaning a multi layered pathfinding algorithm due to the added complexity of the graph on top of the grid, I knew it was best to separate each part of this process. At first I handled finding all the valid paths in my main pathfinding algorithm, but as more and more functionality was added to children (and grandchild later), I eventually in v3 moved this to be a method of the Graph class, returning a 2D vec of nodes traveled.

Now that we know all the paths we must follow, we need to find the cheapest one. We're now using the findCheapestPath function. This function went through the most iterations throughout the assignment as I expanded during each version (and extra credit). At a high level, the function forks a child process, one child process for each valid path. Then a grandchild is called for each pair of nodes that we'll call a "subpath". It needs to compute the subpath's cost, and once all children and grandchildren are done, compute the final cost of each path and choose the cheapest. I choose to keep this function as the architecture for the main problem to solve; it handles going through each path, breaking into subpaths, finding the optimal subpath, and finally computing the cost of each path to determine the cheapest.

I explain more about this process below, but now the child processes are done. We're still in the main loop that goes over all the valid paths, but now we've written information to files for both the children and grandchildren. Now I call a function computePathCost, that will, as implied, compute the cost of the current path. It uses two helper functions to read from the child and grandchild text files. It learns what nodes were traveled from the child, and it learns the total cost and positions from the grandchild. This is much like pipes I realized since we're reading and writing to files in a very similar way. It constructs a LowestCost struct with this information. Through each iteration, if it finds a lower cost, it updates the struct that will be returned to be that one.

That struct is returned and a final function is called to read the struct and handle writing the information to a file in the format requested. That is the end of the program in a general sense. The next section I'll discuss how the child and grandchildren work in detail, as that's the bulk of the code.

Forking Details

Version 1 was quite simple because all I had to do was fork a child process and write out the nodes traveled. We weren't concerned about subpaths or cheapest paths yet. Then in version 2, we needed to implement finding a subpath, but we only needed to output the pairs of the nodes to a file. Naturally, the functions surrounding his behavior changed a lot throughout the 3 versions. Now at version 3, I added a few more functions to break this process down further. In the findCheapestPath function (the main function), we first fork a child process for each valid path. That means we pass a vector of nodes to the child, and the child first creates a scrap file and writes the nodes to it. It will need to know this information later. Then each child will fork a grandchild process for each pair of nodes.

For each pair of nodes, the grandchild will call a function called findCheapestSubpath (now known as the grandchild function). In version 3, we define padding around the search area of the 2 nodes we're looking at, so I compute this first. Now we have a boundary to search along the cost grid to find the optimal path from Node A to B, optimal meaning summing the cost of the positions we travel along the grid for each pair of nodes, so each grandchild will compute a cost and write it to their grandchild text file alongside the positions is travels.
Since I went for the extra credit that requires an optimal path from Node A to B, I implemented Dijkstra's A* algorithm (source in README). This is a separate function that the grandchild function calls. It uses a priority queue to keep track of the cells visited and its own cost matrix that it will dynamically update to compute the cheapest path. It gets the first cell from the priority queue (initialized with the starting node) and checks if we've reached the end. If not, it checks the 8 adjacent cells to find the best place to move next. If moving to a new cell results in a lower cost than previously recorded, it updates the cost matrix and predecessor vector of the new cell to the current cell. Once the destination is reached, we can reconstruct the path it took from the predecessor's vec. We update a reference to the path, which now stores the positions as pairs to get from Node A to B and returns the computed cost. As mentioned previously, all this information is written to the grandchild text file, and now the grandchild is done.

The child must wait for all its grandchildren to complete before exiting, which is crucial or else we’ll waste system resources because the OS might not know to clean up those resources since the process is done, which can result in memory leaks, processes without a parent, etc. From a programmatic standpoint, it’s also crucial to call exit at the exact time needed. Too early and processes won’t run correctly and may stop other processes from being created, such as in this case, and too late and each child process might start creating processes of its own and going through the rest of the code causing many, many issues.

I ran into a problem when trying to compute the cost of a child process sometimes before the child was finished, resulting in only a part of the program being run and weird issues when trying to read from the grandchild’s files. I didn’t realize the code I was trying to run wasn’t being run by the child. All in all, I got a bit mixed up trying to have the child do too many things without waiting for all of them to finish.

After this point, it just computes each path’s cost as all children have finished by this point. I discuss how the cost is computed in the previous section. The only other part of this is the caching implemented for extra credit.

Caching Implementation

Before moving on to discussing computing the path cost and final output, for the extra credit, I also implemented a caching solution to avoid recomputing the subpath cost from Node A to B. I knew this was a dynamic programming problem and memoization would be the solution. We needed to avoid redundant calculations, so a cache seemed like a reasonable solution. I would define a cache implemented as a hashmap storing the path's cost and positions traveled–everything we got from the A* function.

Since I haven't taken CSC 440 yet, I haven't yeti put dynamic programming into practice, so I had to learn how to create my own hashing function since we're working with, bear with me, keys that are a pair of pair of ints (unique start and end positions) and values that are a pair of floats and a vector of pairs (the cost and a vec of positions). A simple hashmap can't handle pairs, so I used multiple tutorials (sources in README) to understand how to do this.

The hashing function PairHash works by combining the hash values of the two elements within a pair. For a basic pair (T1, T2), PairHash first computes the individual hash values of p.first and p.second. These two hash values are then combined with a bitwise XOR and left shift: h1 ^ (h2 << 1). I found this technique is often used to reduce the risk of hash collisions. The left shift ensures that hash values for (a, b) and (b, a) will likely differ, which is useful when key order matters.

As for the values, which are nested pairs ((T1, T2), (T3, T4)), PairHash recursively calls itself for each inner pair, producing two hash values (one for each inner pair) and then combines them using the same bitwise operation.

It took a while to get that all working, but the last step was something I also found in research: handling race conditions with the cache. Since the cache will be used by many different processes at once they could both attempt to write to the same cache entry or retrieve incomplete data, leading to inconsistencies. I had to find the best way to handle this. It turns out that a "mutex" or mutual exclusion primitive can be used here. When a process reads or writes to the cache it first locks the cache. This lock ensures that the process has exclusive access to the cache while the lock is held. Other processes attempting to lock the cache will be paused, waiting until the lock is released.

With the mutex locked, the process safely checks if the desired subpath is already in cache. If it finds the cached data, it retrieves the result and writes it to the output file without releasing the lock, ensuring no other process can alter the cache while it's reading. If the subpath isn't in the cache, the process releases the lock. This is because the process will be busy running the A* function, allowing other processes to use the cache in the meantime. After computing the subpath, the thread locks the cache once more to add the newly computed subpath to the cache. The lock only remains in scope for as long as the grandchild function is running.

So that's the whole implementation for the caching; I'm really happy I was able to get this working.

Testing Implementation

The last major component of the assignment was testing. I knew I needed to do a lot of testing given how many moving parts of the program there were. I decided on creating macros to handle writing output to the console and to files. I didn't want to make the code messy by defining ifdef DEBUG everywhere, so the macros only define the functions when the flag is set when the program is compiled. This is something else I got to learn about when doing the assignment and took a bit of tweaking to get right since I was worried about the overhead of calling so many functions albeit empty ones where the DEBUG macro isn't active. I believe the compiler will optimize this since it sees the function as completely empty. That's what the DEBUG_CONSOLE and DEBUG_FILE function calls are throughout my code.

During multiple points in my code I have functions that write to files. When the DEBUG flag is set by writing -DDEBUG in the compile command. It creates files to output the nodes and edges as well as two files to output nodes in a tree format for all valid paths as well as a tree format for ALL PATHS without the 3 to 5 condition. Naturally, this would take a long time on larger graphs, but that is why this is in debug mode only. I found my (production) code to be very fast even with graphs that are 1024x1024 and 100 nodes.

There's also debug files for how it found the valid paths and debug files for running the A* algorithm. The nodes and edges text field are used by a python program I made using matplotlib to visualize the cost grid with the graph overlaid on top of it to make sure both were being created and travered correctly. I used the tree.txt and tree_valid.txt in another python program to visualize in a tree structure all the paths it found and which ones it thought were valid.

I then used the rest of my debug files to make sure my algorithms were running correctly. I also have extra grid examples for edge cases in the DataSet2 folder. You'll find a few more examples I used for testing in README as well.

All in all, this was a long yet extremely fun and educational assignment due to brushing up on so many areas of Computer Science.