Project Evaluation

Damanvir Singh | Munib Syed

Our final project consisted of a working indoor navigation application which could return the shortest/optimal path given a starting and finishing location. Through the use of A\* (path finding algorithm), we were able to return this short path extremely quickly even on large data sets. The map example submitted with our navigation application consisted of over 2300 nodes, which A\* ran smoothly. Therefore, the navigation part of our application was achieved successfully. The mapping part of our application however, while we had it working perfectly, we simply ran out of time to integrate it into the navigation application. We implemented a feature where the user can upload any image from their local directory and once added, a gem called Tileup would convert the image into tiles ready for the mapping application to take the data. However, since our mapping application was not integrated with the navigation application, we could not take the tile data and feed it to leaflet.js (mapping library) to show a map. The next plan was to use a computer vision algorithm called ray tracing and have it read a map and identify all the walkable space avoiding all obstacles on the map. This code was working perfectly and the map example given with this submission, all the node data for A\* was executed by ray tracing. However, since we ran out of time to integrate the mapping library, this feature could not be automated. Therefore, in this project, we achieved everything we promised and all components of our application were working, we just could not integrate the mapping application with the navigation application together.

Fulfillment of original demands:

Indoor navigation (completed)

Calculating optimal/near-optimal paths using a sufficient path finding algorithm like A\* or Concurrent Dijkstra’s (completed using A\*)

Eraser tool or fill tool to account for unwanted or wanted wireframe obstacles. (not implemented. An eraser tool would require not only clickable nodes, but an ability to drag and select multiple nodes for removal)

Floor plan reading and processing (completed using ray-tracing algorithm)

IPS (if included) using Wi-Fi triangulation (not included in the assignment. It would have greatly increased the scope of the project to unmanageable levels.)

Universal application (completed)

Web application (completed)

User manual (completed)

Deviations:

-No IPS

-No multi-level floors

-No option to define specific nodes as destinations

-No eraser tool

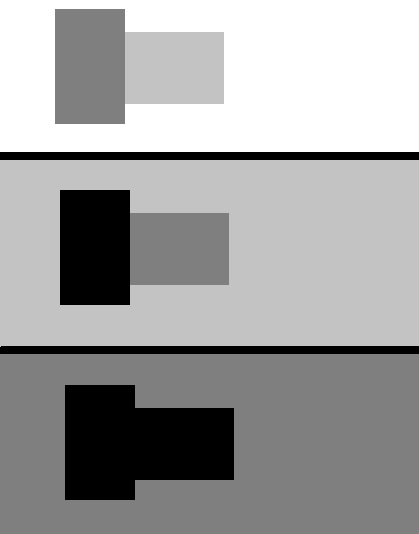
Deviations from timeline:

|  |  |  |
| --- | --- | --- |
| **Deliverables** | **Duration** | **Completion Date** |
| Figure out how to generate a grid of nodes, and create an algorithm that can remove/ignore the nodes within obstacles (using coordinates of corners) to create a walkable space. Make sure we can deal with some simple obstacles/shapes, at the minimum. Test out different mapping libraries to serve our purposes. | 3 Weeks | End Week 5  **Test out different mapping libraries to serve our purposes:** (By Week 2 we had settled on OpenCV as the best image analysis library and Leaflet as the best mapping library for our purposes. Hindered somewhat by installation difficulties).  **Create an algorithm that can remove/ignore the nodes within obstacles (using coordinates of corners) to create a walkable space:**  (Was able to detect all obstacles on a map using Ray-Tracing by Week 4. Wasted time by experimenting with corner-detection, which did not end up being a workable method for detecting obstacles)  **Figure out how to generate a grid of nodes:**  (Was able to generate a grid of nodes by Week 6) |
| Research into more path finding algorithms in order to determine which serves our  purposes the best while still being efficient (Dijkstra’s and A\*).  Work on mapping libraries, and make sure custom maps can be used and manipulated. | 1 Week | End Week 5  **Research into more path finding algorithms:**  (Settled on A\* by Week 7. Wasted some time while researching  Concurrent Dijkstra’s, only to realize that it was not feasible to use with our computer hardware)  **Custom maps:**  (Done by Week 13. Required a lot of research and use of existing software, such as the Tileup Ruby gem) |
| Have a prototype for our application working, which will be able to read in map data (floor maps), place nodes and edges on it while avoiding simple-shaped obstacle (squares and rectangles).  Have an efficient path finding algorithm working.  Be able to use custom maps on the mapping library | 1 Week | End Week 6  **Have a prototype for our application working, which will be able to read in map data (floor maps), place nodes and edges on it while avoiding simple-shaped obstacle (squares and rectangles):**  (By Week 7, this was completed. Any shaped obstacle could be detected at this point, which is better than what we hoped for).  **Have an efficient path finding algorithm working:**  (the basics of A\* were working by Week 11. A\* did not become fully integrated with the rest of the project until Week 16.)  **Be able to use custom maps on the mapping library**  (done by Week 13) |
| Work on complex obstacle detection and complex map reading.Work on algorithm efficiency and be able to handle large amounts of map data.  Work on making the path finding application work with multiple floors in a building. Add more features to path finding application. Work on the application’s cross-platform ability. | 4 Weeks | End Week 10  **Work on complex obstacle detection and complex map reading:**  Began theorizing on this in Week 10, however, most of this was never implemented.  **Add more features to path finding application:**  Many new features were theorized Week 10-12 (changing contrast to compensate for juxtaposing colours, dealing with narrow obstacles), but none were implemented.  **Work on algorithm efficiency and be able to handle large amounts of map data:**  The skeleton of A\* was working by Week 11, which would account for algorithm efficiency.  **Work on making the path finding application work with multiple floors in a building:**  Never implemented.  **Work on the application’s cross-platform ability:**  Done by Week 16. The application did not become multi-browser supportable by this time. |
| Research into IPS and test with different resources and methods. | 1 Week | End Week 11  IPS removed from project. |
| Build a IPS | 4 Weeks | End Week 15  IPS removed from project. |
| Integrate the IPS into the main application.  Work on the documentation | 1 Week | End Week 16  IPS removed from project. |

Challenges:

-Discovering the best way to detect obstacles. We knew about the ray-tracing algorithm, but we ignored it assuming that it would be far too slow for us to use, and we reserved it as a backup method for detecting obstacles on a map. We experimented a lot with Harris corner detection to detect squares and the area inside them and Barycentric coordinates for detecting triangles along with the area inside them, but with no progress. Having a separate obstacle detection method for every shape type was problematic; how would the program know which method to use if it didn’t know what shape had been encountered?  We realized the unworkable self-referential nature of these solutions and tried to implement ray-tracing instead. We discovered that ray-tracing was far easier to implement and gave much quicker results than we expected. We quickly abandoned other methods and found that ray-tracing was the best method for the job.

-Discovering the limitations of map-reading. We knew that there would be certain situations where ray-tracing would deliver faulty results, and we worked on hypothetical ways to counter this. One example is when there are very narrow obstacles on a map. The ray-tracing algorithm searches for obstacles using a certain “jump” value, because searching each and every pixel on the map would be far too inefficient. The problem lies when an obstacle begins and ends within n number of pixels (with n being the jump value). This obstacle will never be detected and the path-finding algorithm may trace a path straight through a wall. We devised a method that should have fixed this; if two or more edges are detected between two points that are n number of pixels away from one another, then a second, more-thorough obstacles search will be implemented between the two points, which would result in the narrow obstacle being accounted for. However, this solution would only work with Dijkstra, where a strict grid structure is not necessary. Our implementation of A\* works on the assumption of a strict grid structure, and thus this solution (which, would create nodes between nodes and break the grid structure) could not be implemented with A\* and had to be discarded. Another limitation of map-reading is an obstacle with a sharp colour change within it. Because a sharp colour change would be perceived as an ‘edge’ to the ray-tracing algorithm, this would result in many points on the map being incorrectly perceived as obstacles, or obstacles being perceived as walkable space. The solution for this was to add an option to change the contrast of an image. If an image is darkened sufficiently, the two juxtaposing colours will eventually be perceived as the same colour; essentially fixing the problem. We worked for some time on a slider for OpenCV, which could be used to adjust the contrast of an image. There would need to be some kind of visual feedback to the user in order for them to tell when this fix has ‘worked’, so ideally, ray-tracing should be run continuously while the contrast is being modified. However, it proved difficult to both darken the image and continuously run ray-tracing at the same time. This also raised questions about the efficiency of such visual feedback. Ray-tracing usually takes at least few seconds to scan an image, so if ray-tracing is called every time the slider changes value, there was the potential for serious lag. Even though it is useful in theory, it seemed that this feature was not easily implementable without impacting the performance of the application as whole, so this idea was abandoned.



- Dealing with wireframe obstacles. Because these obstacles have blank space in their centre, this space will incorrectly be perceived as walkable space. Two possible solutions were hypothesized for this challenge. One was a fill tool. A user could use the tool to fill the blank space within an obstacle to make it match its outline colour, making it look like one solid obstacle which will not be incorrectly reading by the ray-tracing algorithm. This comes with the sacrifice of user convenience, as there may be many wireframe obstacles that users must fill out. Another proposed solution was an eraser tool. After the algorithm has incorrectly placed nodes, the user can use this tool to delete any unwanted nodes. we were leaning towards this method but we could not find any way to successfully record a user’s drag movements, in order to clear many nodes at once. There was a method for flood-filling an image using OpenCV but we did not get time to implement it.

-   IPS. Never implemented due to time and a wish to avoid a large scope. Challenges include lack of knowledge about Wi-Fi systems and the need for advanced networking and mathematical knowledge.

-Optimising the jump-value/running time. To make sure running time stayed underneath a certain threshold, an automatically adjusting jump value was hypothesized. The jump value would increase as image size increased, creating fewer nodes and thus increasing running time for both ray-tracing and A\* pathfinding. However, this was never implemented and a static jump value of 20 was used. This was because the running time ray-tracing grew in an unpredictable way. In order to keep the running time underneath a certain value, it is essential to know how the running time of ray-tracing increases as the image size increases. However, we found it too difficult to discover the growth rate of the algorithm in the way that we implemented it.

-Mapping library. Being able to use custom images on a mapping library. This is because the mapping library was officially designed to work off the database the mapping library came with. This was solved with a gem called Tileup. With Tileup we could convert an image into multiple tiles with various zoom levels and by feeding these tiles into Leaflet as map data we were able to achieve what we wanted. This however, created more issues. All functions needed to manipulate the map, were configured to the original map data by default. This mean that while we could display a map, we could not use it in terms of placing markers, nodes, paths or even proper zoom levels. This issue was solved with further research into the extensive leaflet.js API, where we found limited but useful ways to import external map data while still being able to use it. This required every single absolute value being fed into leaflet, to be converted from pixel coordinates to latitude and longitude coordinates. Once implemented correctly, the process became easy, however this required us to re-structure our whole leaflet side of the application as our old values/data could not be read.

-Displaying the shortest path. Leaflet needed to have a list of functions (map.unproject)  to create a Polyline layer in order to display the shortest path. However, working with different programming languages and different data types between files, this process became more complex. Processing data (a list of the shortest path) and returning it in a format that Leaflet could interpret was a challenge. Our previous solution for values was necessary but we needed further formatting of the data. This issue was overcome through further processing of the data in JavaScript using JSON through AJAX. The solution consisted of an AJAXIAN request being made from the server for a starting location and ending location. This data was taken and then sent to Flask through AJAX. Flask would then feed that data into a python function which would run A\* and return the shortest path. That shortest path was then converted into a JSON object and returned to the server through AJAX once again, and this is how we were able to display the shortest path on the map.

-Executing shell commands through Python. This proved difficult because there was a lack of consistency between inputs required for certain functions but the biggest issue was changing shell directories. For example, some functions required an absolute file path in order to work while others didn’t. Differentiating between these functions proved challenging and slowed progress because we had to use trial and error on all of them.

- Integrating OpenCV with Flask. This was an  
extremely challenging and risky part of our project as the integration between  
the two major components (Computer Vision and Server Side) could only be done  
at the end once each individual components was in working condition. Since  
this project consisted of extensive research and continuous restructuring,  
integrating the two components at the start would have slowed the pace of this  
project exponentially. If one component’s structure was changed and the other  
component was being built in order to work with the first component, then both  
of them would have had to been restructured and built again, which would have cost us time.  
For example, a mapping library was being used to which the computer vision code   
was connected to. If however, the mapping library was to be  
changed, then so would the computer vision component as it was built to work with that mapping library only. The most efficient way to integrate the two components was to build the two components individually but keeping the other component’s structure in mind.

For example, if a mapping library and a computer vision library were being used, they would be worked on individually but the data that was later to be transferred between the two libraries, that was kept in consideration, hence our code around the data we needed to transfer. However, as mentioned before, our project went through necessary constant restructuring which cost us time and due to this we were not able to integrate the two component.