Vision for Autonomous Robots
Lab Project 3: Mapping & Route Planning

Wouter Bant (13176676)
Angelo Broere (13168215)
Gidi Ozery (14643618)
Valeria Sepicacchi (15187853)

Due: December 15th, 2024

1 Introduction

In this assignment, we work with VisualSFM to create sparse reconstructions of different mazes. Using kd-trees we filter out outlier points and with this same datastructure we also show a way of making sparsely reconstructed areas more densely occupied with points by leveraging the straightness of walls.

Further, we chose to do the second and third bonus parts. For this, we measure the location of all Aruco tags in the maze and use the localization and pose estimation algorithm from lab 2.

2 Session 1: Structure-from-Motion

2.1 Part 1

2.1.1 Keysteps SfM

1. Feature detection and description: Extract keypoints and corresponding descriptors from all views. For this, you could use SIFT as these are detected and have similar descriptors under varying conditions, which is desired for the next step.

2. Feature matching:

   - For each descriptor look for similar descriptors in other images.
   - Estimate the fundamental matrix with RANSAC between each image pair.
   - Using the fundamental matrix, filter out the matches inconsistent with the geometry or camera position. The result is a connectivity graph

3. Incremental reconstruction:

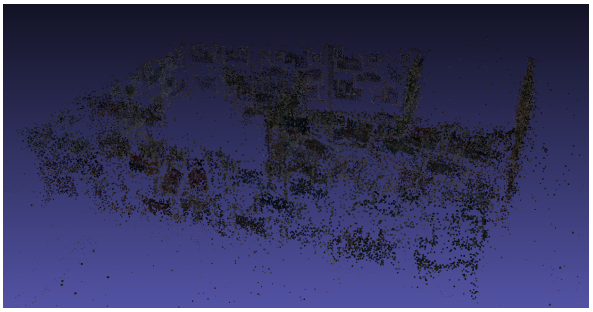
   - Start with an initial pair of images with good feature matches and a wide baseline to reconstruct an initial 3D point cloud. Add new views incrementally with the next steps.
   - Decompose the fundamental matrix between two frames into relative translation/rotation.
   - Using the found 3D to 2D correspondences do camera pose estimation with PnP or DLT.
   - Use the corresponding points in multiple views and the estimated camera poses to solve for new 3D point positions using triangulation.
   - Bundle adjustment (global optimization): re-project the 3D points back onto the image planes and minimize the re-projection error by adjusting (1) the location of the points, (2) intrinsic matrices, (3) camera positions and poses, and (4) camera distortion (this can be done efficiently with the Levenberg-Marquardt algorithm).

4. Sparse point cloud: The estimated 3D locations of the features make the sparse point cloud. To get a dense reconstruction Multi-View Stereo can be used.

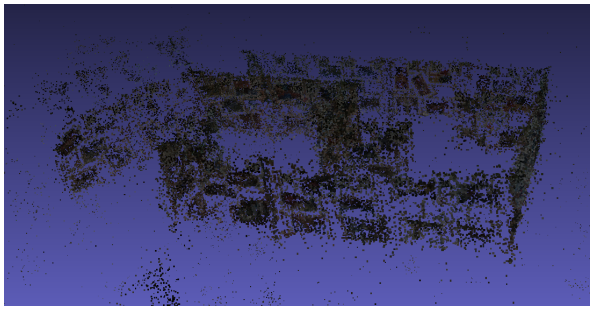
2.1.2 Weakness of Reconstruction

For this part, we apply VisualSFM on all frames extracted from the provided video and do an exhaustive search for matches between frames. These results are shown in Figure 1. Figure 1a shows that the camera positions are estimated quite well even the first view frames where the robot is not on the ground are accurately captured. However, figures 1b and 1c show varying densities of points across the maze. Some pictures in the maze can almost be seen solely from the points, but we see a gap in places without images. This is because in these areas no features are found. Further, we see that walls that were only a short time in the video have a lower density of points (as seen in Figure 1b). This is because some detected features will likely not be used as there are insufficient frames with strong enough matches.

(a) Sparse point cloud with estimated camera positions and poses.



(b) Sparsity of point cloud example 1



(c) Sparsity of point cloud example 2

Figure 1: Initial reconstruction of the maze from the provided video.

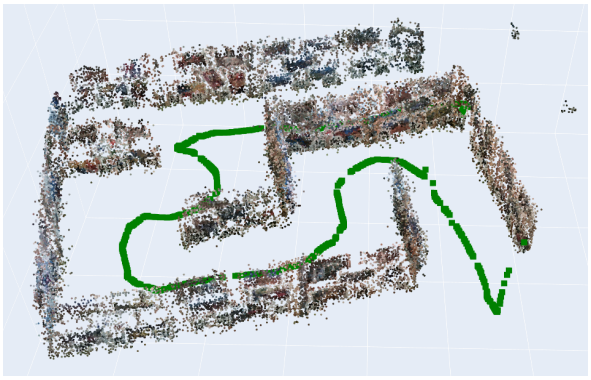
2.1.3 Improvement

To address wall sparsity, we adopt the following method: For each detected point, calculate the average position and color of its neighbors within a given radius and propose this as a new point. If fewer than k points exist within a smaller radius of this location, accept the new point. This leverages the inductive bias that points on the same wall lie on a plane, ensuring the averaged position remains on the wall. By adding points only in sparsely populated areas, we focus on under-reconstructed regions. The implementation is as follows:

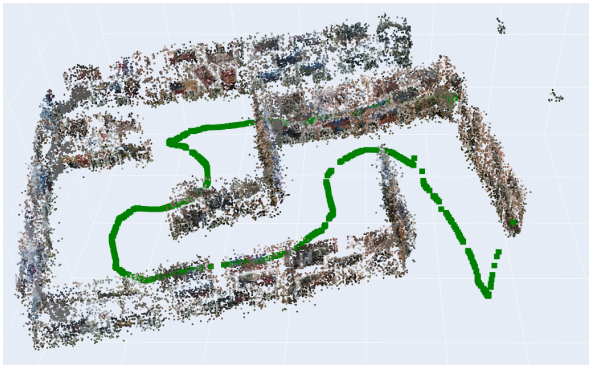
```
for i in point_indices:
    point = points[i]
    neighbors = tree.query_ball_point(point["coordinates"], average_radius)
    # Point proposal: calculate the average of neighboring point coordinates
    neighbor_coords = point_coords[neighbors]
    new_coord = np.mean(neighbor_coords, axis=0)
    # Sparsity check: accept if point isn't too close to many existing points
    if len(tree.query_ball_point(new_coord, near_radius)) < k:
        new_points.append(new_coord)
        new_rgb = np.mean([points[j]["rgb"] for j in neighbors], axis=0)
        new_colors.append(new_rgb)
```

In Figure 2 we display the densification results for various values of k. As expected the higher the value of k the more points are added, this is because not only sparse areas will get more points but also semi-dense areas (as proposed points get more easily accepted). Still, for $k = 20$ and $k = 30$, there are

some sparser areas but these are significantly denser than before. A slight limitation of this method is that, although a higher percentage of points gets accepted in sparse areas, fewer points get proposed as there are just fewer points in these areas. A solution to this is to repeat this process multiple times with the newly added points (as in every iteration there will be at least as many proposals in every area as previously), however, this makes the process a lot more expensive as the kd-tree needs to be built before every iteration and this is an $O(N \log(N))$ time operation, where N represent the growing number of points. Also, the predicted colors will become more uniform over iterations as already can be seen in some areas in Figure 2.

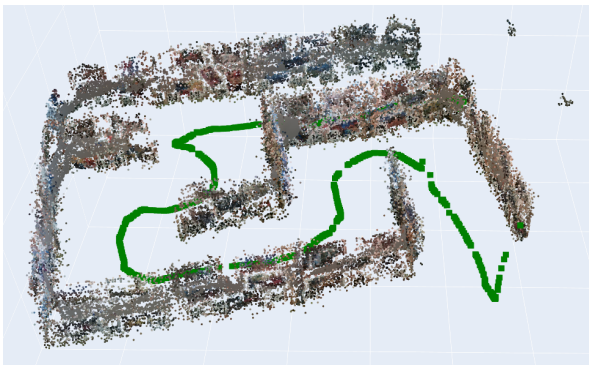


(a) Original reconstruction.



(b) $k = 10$.

(c) $k = 20$.



(d) $k = 50$.

Figure 2: Making sparsely reconstructed areas more dense with different values of k.

2.2 Part 2

2.2.1 Topic List for Video Recording

We used the $/rae/right/image\_raw$ topic and rectified the incoming frames with the camera calibration found for assignment 1. Further we limited the fps to 20.

2.2.2 Collection New Maze

2.2.3 Offline SfM

Also for this part, we choose to use visualSFM again. The sparse reconstruction results are visualized in Figure 4 and the results after Multi-View Stereo in Figure 5. We had to use the uncompressed images with sufficient overlap between consecutive frames to get this good of a reconstruction. Also, we had to take the video in the middle of the day when the lighting in the lab was the best.

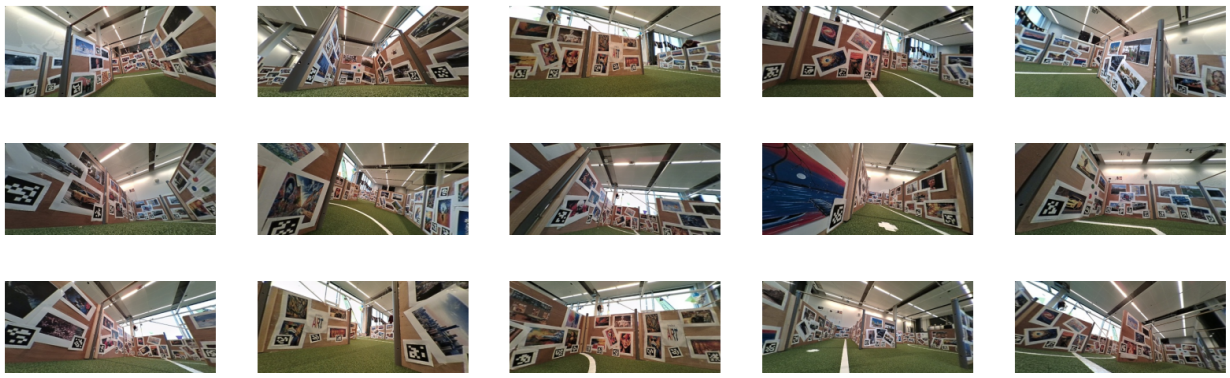


Figure 3: Randomly selected frames of the week 2 maze.

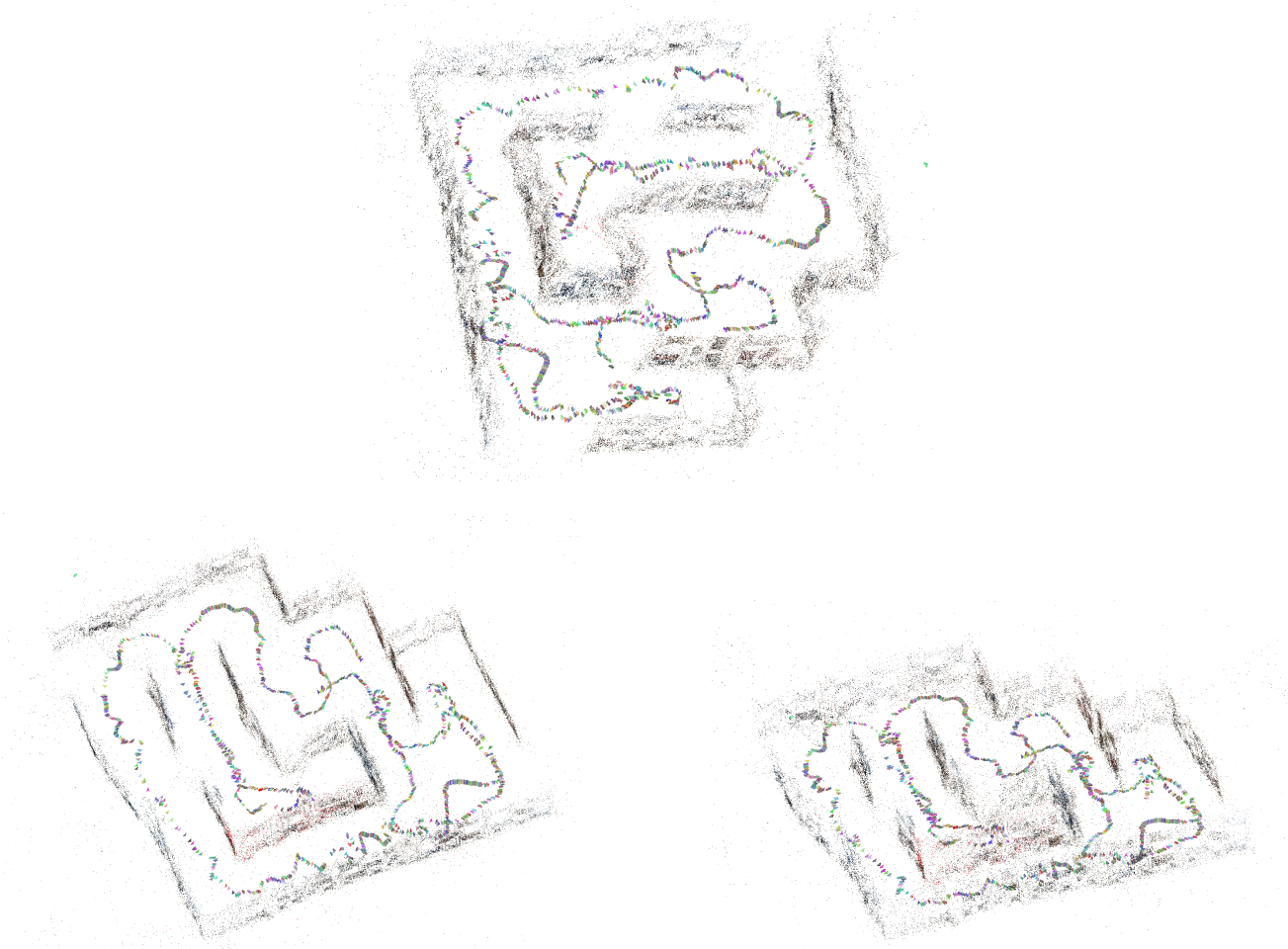
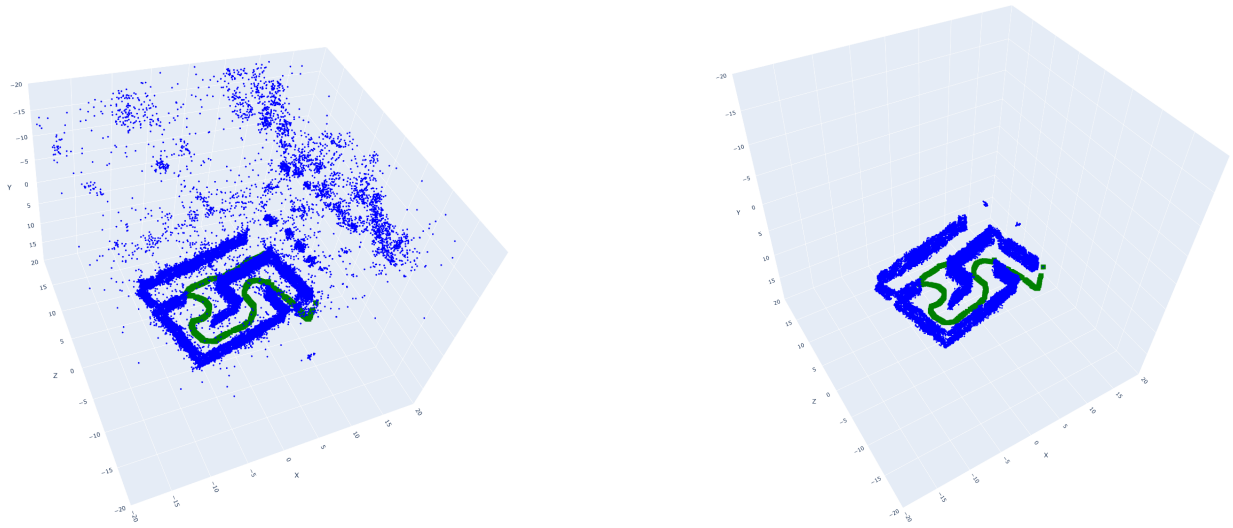


Figure 4: Sparse reconstruction results.

Figure 5: Results after Multi-View Stereo.

3 Session 2: Mapping

3.1 Removing keypoints far away from the maze



(a) Before removing outliers.

(b) After removing outliers.

Figure 6: Before and after removing outliers with a kd-tree.

From the plot with outliers (Figure 6a) we can see that most outliers are not near many other points compared to the points that make up the maze. This makes sense as the maze has many pictures of highly textured scenes, whereas most outliers are particular spots picked up on the ceiling. Therefore, our solution for removing the outliers is to remove the points with few nearby neighbors.

A brute-force approach, which computes distances between all pairs of points, has a poor time complexity of $O(N^2)$, where N is the number of points. Instead, we use a kd-tree, which can be built in $O(N \log(N))$ time and performs neighbor queries in approximately $O(\log(N) + m)$ time per point, where m is the number of nearby points. The overall expected complexity is $O(N(\log(N) + m))$, as $m \ll N$ in most cases, this scales much better than the brute-force approach.

This is implemented as:

```
point_coords = np.array([point["coordinates"] for point in points])
tree = KDTree(point_coords)
filtered_indices = [
    i for i, coord in enumerate(point_coords)
    if len(tree.query_ball_point(coord, radius)) >= neighbor_threshold
]
filtered_points = point_coords[filtered_indices]
```

Figure 6b shows the effectiveness of this method: almost all outliers are removed while the points making up the maze are preserved.

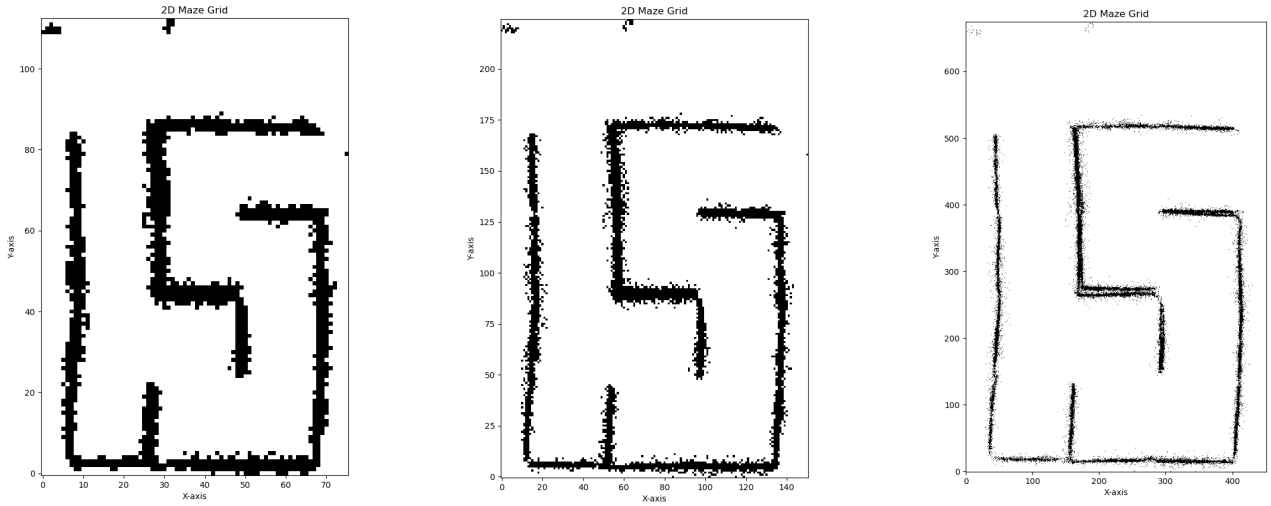
3.2 Creating Topology Mapping

We project the sparse point cloud onto the ground with approximate cell decomposition to get a 2D representation of the maze. To be able to do this we need to discretize the available space, allowing for many squares will make search algorithms more computationally expensive but allowing too few pixels will lead to less accurate routes and mapping of walls. We visually compare different 2D maps in Figure 7. From this, we see that the more pixels we allow the more accurate the width of the walls becomes. However, depending on the path planning speed required, one can use a lower-resolution map.

It can be seen that the walls still contain some noise. As we know the walls are not actually like this we apply a median filter to remove this noise, this result is visualized in Figure 8

3.3 Discussion and Future Work

So far we have identified two problems and at least partially solved them:



(a) Few pixels. (b) Double the pixels as 7a. (c) Many pixels.

Figure 7: Comparison of the 2D map with different number of pixels.

(a) Few pixels. (b) Double the pixels as 8a. (c) Many pixels.

Figure 8: Comparison of the 2D map with different number of pixels after applying 3x3 median filter.

- The projected maps contain noise as the estimated x and y locations of the points are not always fully accurate. We have removed these points on the grid by applying a 3x3 median filter.
- Using a high-resolution 2D grid can be computationally expensive, therefor we compare how many pixels we need. Using many pixels leads to better estimation of wall width, however, it is not even a bad thing that walls are slightly wider as this makes avoiding them more likely. In the next section, we will display the paths on the maze of Figure 8b.

Three problems we see that could be addressed in the future:

- The wall on the left in our maps is not fully straight reconstructed. We checked with the actual maze and this wall should be straight. One could solve this by estimating lines (walls) of the maze with a line detection method (e.g. Hough transform) combined with a heuristic that keeps the most horizontal/vertical lines. This would also solve the problem of noise.
- There are gaps in the map of the highest-resolution maze after the median filter. This can be solved by extrapolating the found lines of the previous proposed solution until different lines meet. If no intersection is found after some amount of extrapolation one can conclude that this must be the start or end of the maze.
- For route planning not every area has to have the same block size. With adaptive cell decomposition you could (1) save memory and (2) find faster solutions for route planning.

4 Session 3: Route Planning

4.1 Keysteps Dijkstra's algorithm

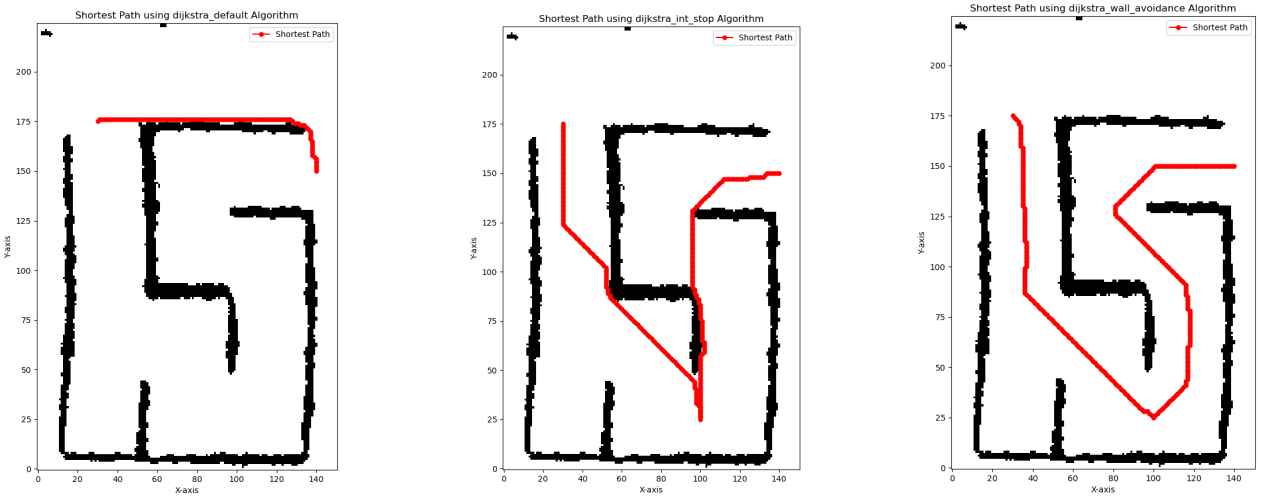
At the end of this section we show our implementation but the keysteps are:

1. Define: grid, start and end location, and possible moves
2. Initialize:
   - Cost datastructure to maximum values (float("inf")).
   - Prev datastructure to store previous locations for path reconstruction after finding a solution.
   - Priority queue with start location and 0 cost.
3. While the priority queue is not empty and the destination isn't found:
   - Pop the top element from the priority queue, if this is the destination, break out this loop.
   - For all possible moves evaluate the cost = current cost + move cost
   - When the cost is lower than what is currently stored replace it, update the prev datastructure, and add the neighbor to the priority queue with associated cost.
4. Reconstruct the path with prev datastructure and return it.

4.2 Implementation Dijkstra's Algorithm

We input the start and end coordinates and see that the default implementation of Dijkstra's algorithm (Figure 9a) finds the shortest path, but this doesn't go through the maze. In Figure 9b we solve this by first going to an intermediate location before the final destination, we do this by calling the algorithm from before two times and merging the paths.

However, we can see that the resulting path is, in fact, the shortest but goes close past walls, failing to take into account the width of the robot as this is represented with a single grid. To address this we implement a version of Dijkstra's algorithm that gives high costs to locations close to a wall. To be specific, we add to the normal cost the factor $50/(1 + ||new\,location - closest\,wall\,point||_2^2)$, which we experimentally found to provide good routes. The resulting path is visualized in Figure 9c and is a reasonable path for the robot to follow.



(a) Default Dijkstra. (b) 9a + intermediate stop. (c) 9b + wall distance cost.

Figure 9: Comparison of planned paths.

We now present the code to obtain the various figures:

```python
class RoutePlanning:
    @staticmethod
    def calculate_wall_distance_cost(grid):
        binary_grid = grid.astype(bool)
        wall_distance = ndimage.distance_transform_edt(~binary_grid)
        return wall_distance

    @staticmethod
    def dijkstra(grid, start, end, wall_weight=0.0):
        wall_distances = RoutePlanning.calculate_wall_distance_cost(grid)
        directions = [
            (-1, 0), (1, 0), (0, -1), (0, 1),  # cardinal moves
            (-1, -1), (-1, 1), (1, -1), (1, 1),  # diagonal moves
        ]
        rows, cols = grid.shape
        dist = np.full_like(grid, float("inf"), dtype=float)
        dist[start[1], start[0]] = 0  # Start distance is 0
        prev = np.full_like(grid, None, dtype=object)
        pq = [(0, start)]  # Starting point

        while pq:
            current_dist, (x, y) = heapq.heappop(pq)
            if (x, y) == end:  # At final destination
                break
            for dx, dy in directions:
                nx, ny = x + dx, y + dy
                # Calculate move cost (diagonal moves sqrt(2), cardinal moves 1)
                move_cost = np.sqrt(2) if dx != 0 and dy != 0 else 1
                if 0 <= nx < cols and 0 <= ny < rows and grid[ny, nx] == 0:
                    # Get wall distance cost for this cell
                    wall_distance_cost = wall_weight * (
                        1 / (wall_distances[ny, nx] + 1)
                    )
                    # Total cost: base move cost + wall distance penalty
                    new_dist = current_dist + move_cost + wall_distance_cost
                    if new_dist < dist[ny, nx]:
                        dist[ny, nx] = new_dist
                        prev[ny, nx] = (x, y)
                        heapq.heappush(pq, (new_dist, (nx, ny)))
        # Reconstruct the path from end to start
        path = []
        curr = end
        while curr != start:
            path.append(curr)
            curr = prev[curr[1], curr[0]]
        path.append(start)
        path.reverse()
        return path

    @staticmethod
    def dijkstra_through_intermediate(grid, start, inter, end, wall_weight=0.0):
        path_start_to_intermediate = RoutePlanning.dijkstra(
            grid, start, inter, wall_weight
        )
        path_intermediate_to_end = RoutePlanning.dijkstra(
            grid, inter, end, wall_weight
        )
        complete_path = path_start_to_intermediate + path_intermediate_to_end
        return complete_path

fig_a = RoutePlanning.dijkstra(grid, start, end, wall_weigth=0)
fig_b = RoutePlanning.dijkstra_through_intermediate(grid, start, inter, end, 0)
fig_c = RoutePlanning.dijkstra_through_intermediate(grid, start, inter, end, 50)
```

5 Bonus

5.1 RAE robot automatically runs the planned route (5%)

We measured the location of all Aruco markers in the maze and used this for localization and pose estimation. This is done in the same way as in lab 2. The location and pose estimation were much more accurate than in the previous assignment, the main reason for this is that many more markers are detected, illustrated in Figure 10.

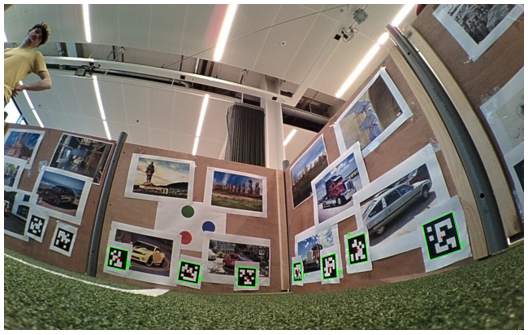


Figure 10: The localization and pose estimation are accurate as many markers are detected.

We specified the start and end locations to compute the path, incorporating wall costs. For this maze, it was unnecessary to input intermediate points, as all shortest paths naturally passed through the maze. At the time of the demonstration, we were dissatisfied with the reconstruction and mapping results of the week 2 maze. As a result, we manually applied filtering and selected the best-fitting lines through the obtained points. The preplanned path on this map is illustrated in Figure 11.

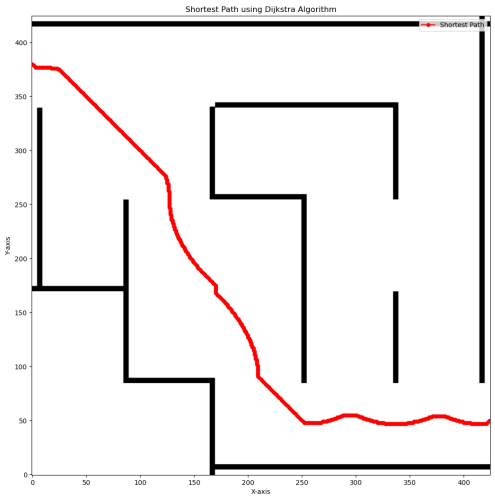


Figure 11: Preplanned path on manually improved mapping of the maze. The zigzagging around $y = 50$ is because of wall avoidance cost and was also observed during the demonstration.

After localization, we determine the point on the path closest to the robot and take the point 15 cm further along the path as the target for the movement command. The angle is calculated based on the robot's estimated location and target point.

5.2 RAE robot automatically moves out from the maze (10%)

When put in a random position we do everything the same except the start point is the result of the localization of the first incoming frame after putting down the robot and starting the node. Figure 12 shows an example of such a path. In the example, the robot was tasked to find a path to the left upper exit. It chose to go down as this leads to a path where the distance to the walls is on average larger. During the demonstration, our solution proved robust against kidnapping the robot halfway through the route and putting it back a couple of meters.

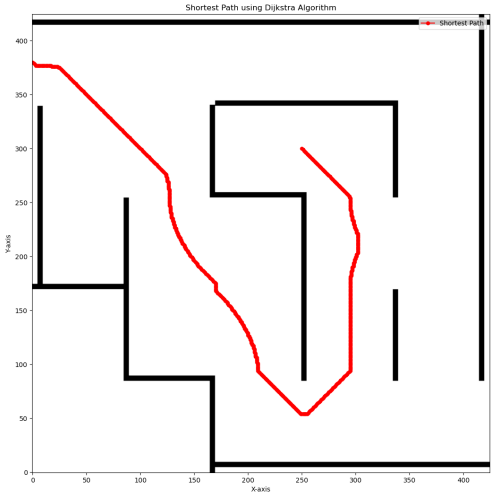


Figure 12: One of the paths that was calculated with the localization result of the first frame after putting down the robot. The robot decided to go to the target via a slightly longer path as this leads to a path where the distance to the walls is on average larger (lower wall avoidance cost).

6 Conclusion

In this assignment, we explored using VisualSFM to create sparse reconstructions of two different mazes. Utilizing kd-trees, we effectively filtered out outlier points and enhanced sparsely reconstructed areas by leveraging the straightness of walls to increase point density. The use of kd-trees significantly improved the scalability of our approach compared to the brute-force method of comparing each feature point with all others.

Additionally, we chose to complete the second and third bonus tasks. For these, we measured the positions of all Aruco tags within the maze and used the localization and pose estimation algorithm from Lab 2. To ensure the robot avoids walls, we incorporated a wall avoidance cost into the distance cost of Dijkstra's algorithm. This approach favors paths that maintain a greater distance from walls over those that closely pass by them.

Thank you for the fun assignments!