NavPal Floor Plan Authoring Tool Preprocessor Summary

Zhiyu Wang
Carnegie Mellon University 15-239

Abstract

NavPal is a smart mobile phone-based personalized navigation aid to enhance the navigation capability and thereby the independence and safety of visually impaired people. It is a joint effort by TechBridgeWorld and r-commerce Lab in CMU[1].

NavPal Floor Plan Authoring Tool preprocessor attempts to detect and extract key features from a standard white-background floor plan image for floor managers to verify and dynamically modify. Key features include walls, doors and texts. Preprocessor is written in python 2.4 and requires OpenCV2 and Python Tesseract installed. Due to time constraints, the authoring tool preprocessor is not completely polished. Current version requires a standard, relatively high-definition floor plan image. Future work also involves improving the efficiency and robustness of object detection.

User Guide

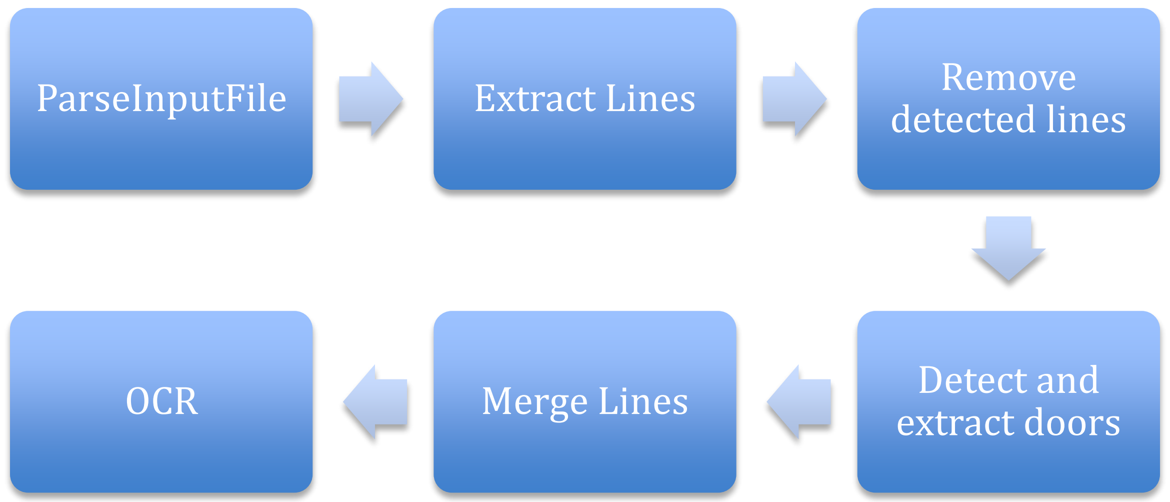
1. Installation: OpenCv2, Python Tesseract , (Python Image Library)
2. Files relevant:
    - ➢ preprocessor.py: main wrapper function for preprocessor
    - ➢ utilities.py : all general utilities function used in preprocessor.py
    - ➢ constant.py : contain all constants used in preprocessor
    - ➢ classes.py : constant all classes used in preprocessor
    - ➢ extractLine.py : contain all helper functions in line extraction
    - ➢ merge.py : contain all helper functions in line merging
    - ➢ OCR.py : contain all helper functions in text detection and extraction
    - ➢ matching.py: contain functions used for template matching of doors

3. How to run preprocessor:
   In terminal, enter "python preprocessor sourcePath destPath dataPath" where
   sourcePath: path to the floor plan image;
   destPath : path to where the preprocessed image is saved;
   dataPath: path to where the extracted objects are saved;

   *To turn off visualization, comment out the 'visualizeLines' function
   in preprocessor.py.

[1] Cited from http://www.cs.cmu.edu/~navpal/

General Flow



- ParseInputFile: Read image from the sourcePath and save it as an image object
- Extract Lines: Extract lines horizontally and vertically respectively.
- Remove detected lines: Removes line segments for door and text detection.
- Merge Lines: Merge lines in close proximity to lessen human authoring work
- OCR: Detect text and extract data.

*OCR, door detection and line merging are in fact three independent processes. I have not explored the possibility of implementing them in parallel.

*Moreover, multiple floor plans can be processed in parallel.

Technical Details

A. Parse Input Files

Current version is using Python Image Library to extract all useful information from the image and save the image as an object defined in classes.py

B. Extract Lines

The preprocessor detect lines by looping through the image pixel array. Starting from a non-background pixel (considering it as the endpoints of a line), it checks whether adjacent pixel (on its right for horizontal line, below it for vertical line) is also non-background pixel. If it is, it extends the 'line' till it finds a background-color pixel. It then considers the background-color pixel as the other endpoint of the line. If the length of the new line segment exceeds minimum length, it saves it into a list.

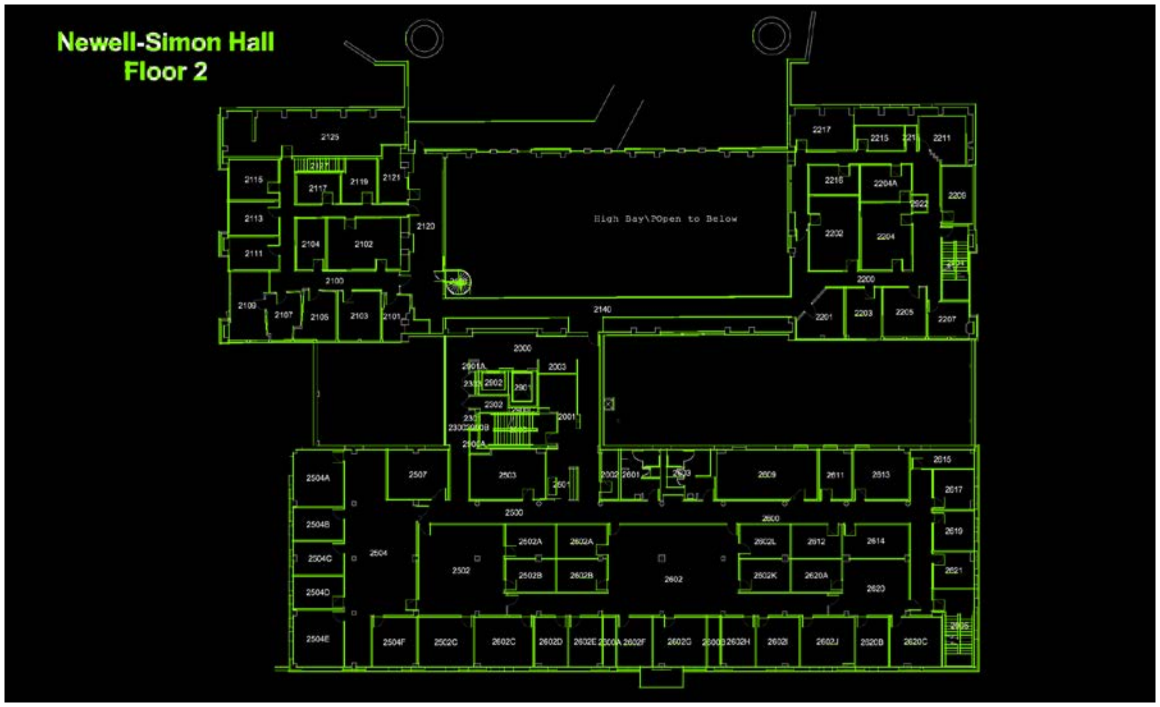


Figure 1: Effect of line extraction

Through testing on Carnegie Mellon University Newell-Simon Hall 2nd Floor, the preprocessor detects and extracts over 95% of the lines. The limitation is that I assume that all walls in the floor plan are either horizontal or vertical.
Future improvement needs to resolve the issue of tilted lines.
However, generally, for standard floor plan, line extraction is decently effective.

C. Extract Doors

The preprocessor detect doors in two phases:

1. Find Contours.
2. Template Matching.
3. Direction detection.

At this stage, we already removed all the lines detected from the floor image in order to have a cleaner image (with less noises) to work on.

1. Find contours:

At this stage, we use Opencv2's findContours function to detect all connected components in the image. We then use pre-set parameters of doors (width and height) to filter out non-door objects.

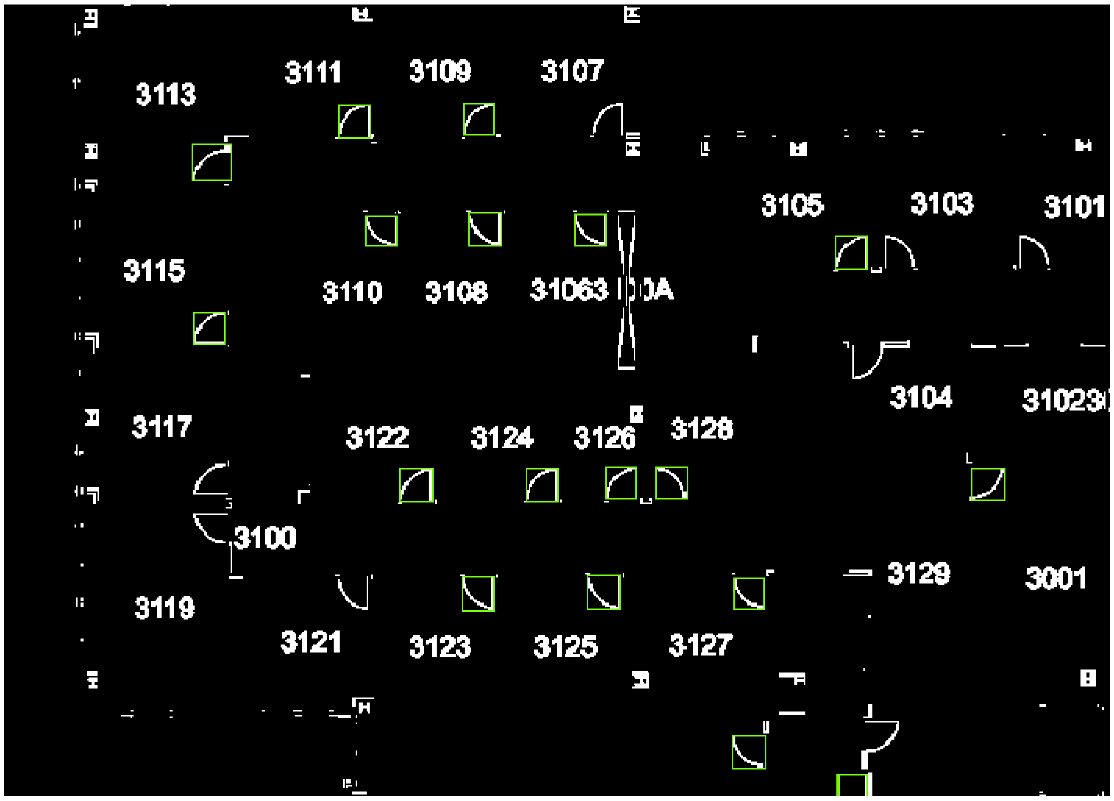


Figure 2: door detection with "Find Contours"

At this stage, the door detection rate is approximately 70-75%. Many doors are missed because the connected components they are in did not pass the parameter filter. That is why the 2nd stage is necessary: Template Matching.

4. **Template Matching**

Once we have a candidate for doors, we then rotate the image by 90, 180, 270 degrees and perform template matching to extract all door candidates in the image. This is implemented by cv2.matchTemplate and cv2.minMaxLoc function from opencv2 API. We then obtain the bounding rectangles of the doors and save them into a list.

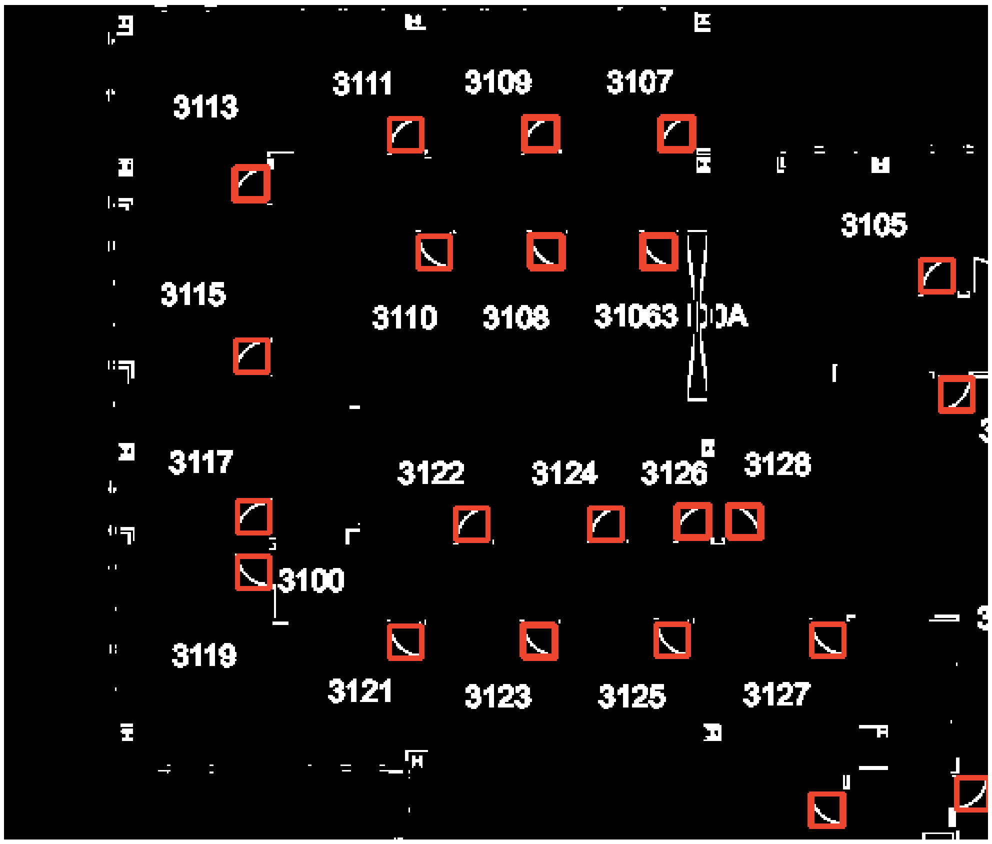


Figure 3: Door detection with ‘Template Matching’

Figure 3 demonstrates the improvement of door detection rate compared to the same area in Figure 2.
The door detection rate at second phase reaches 85-95%. There are still some doors that are not detected. This is partially because the sub-image of some doors is rendered disconnected when lines are removed. Further improvement involves ‘smoothing’ the image using morphological operations followed by template matching.

3. Direction detection

At this stage, we are able to extract approximately 85-90% of the doors and have a list of the bounding rectangles in hand. It remains to determine which edge of the bounding rectangle is the door.

What the preprocessor does now is that for each bounding rectangle, it locates the center of the bounding rectangle in the original image. Then it draws four probe lines in up, right, down, left direction respectively. The direction in which the probe line in that direction intersects no line is the door edge.

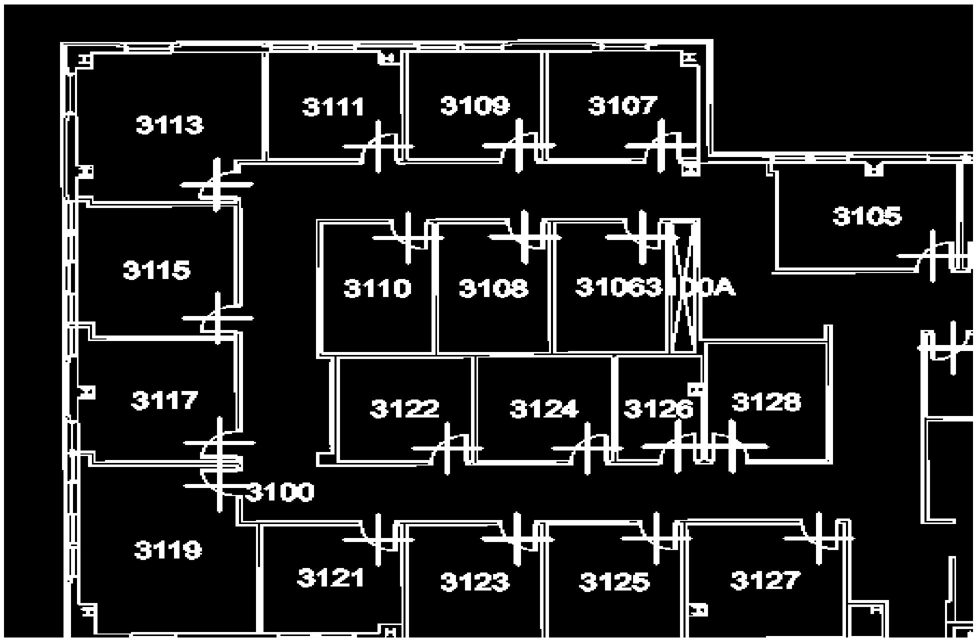


Figure4: Direction Detection

After this stage, we gather all the door lines detected and save them into a data file which will then be used by authoring tool.

D. Merge Lines & Vertices

Despite the high line and door detection rates by the preprocessor, the data gathered by far is not user friendly. This is because most floor plans do not represent a wall by a single line. In majority cases, a single wall is accompanied by many additional lines, which are unnecessary in NavPal's graph representation. In order to ease floor manager's work, the preprocessor merges lines in close proximity such that the floor manager has a cleaner image to work on. This is implemented by checking the distance between every pair of horizontal/vertical lines. If the distance is less than the epsilon threshold, we then merge the two lines into one. The complexity is O(n^2). I feel very uncomfortable with this implementation and will work on improving the efficiency of this process.

Besides line merging, the preprocessor also merge vertices in close proximity. This is implemented in a similar manner as line merging.

E. Extract Texts

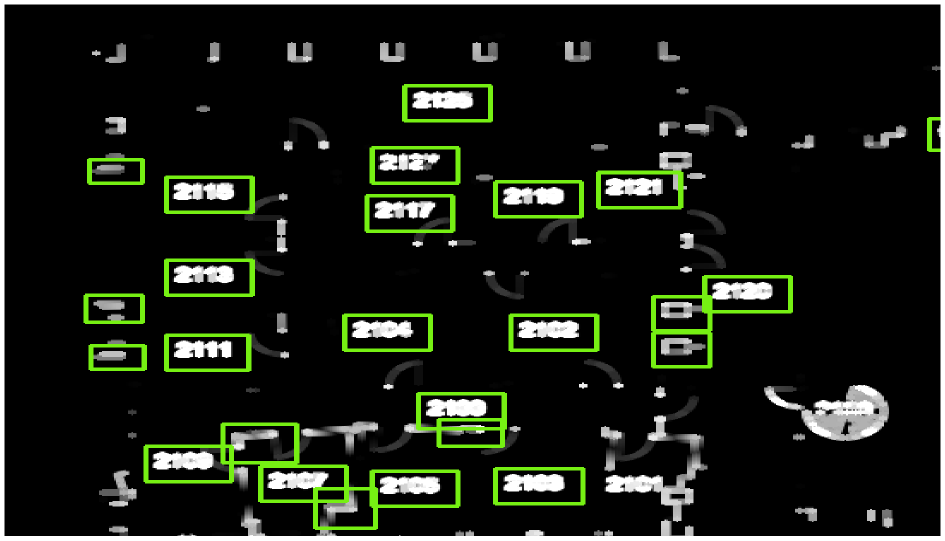
Text extraction is implemented in a similar manner as door detection.

It has three phases:

1. Morphological dilation
2. Find Contours
3. OCR

At this stage, we already removed all the lines detected from the floor plan so that we have a cleaner image to work on.

1. **Morphological dilation**
   The purpose of morphological dilation is to make texts into blocks so that we can identify each block of texts as one connected component.
2. **Find Contours**
   Similar to door detection, we then use cv2.findContours function to identify all connected components and then filter the list of contours with pre-set parameters. At this stage, we obtain the bounding rectangles of all detected text blocks.
3. **OCR**
   At this stage, we have a list of bounding rectangles. We then take the original image and obtain a copy of the sub-image of the original image in the bounding rectangle region. Passing the sub-image as an input to Python Tesseract, which is an optical character recognition API, we can obtain the text data and the confidence level that the text data is thought to be correct.
   We then filter the text data with a threshold confidence level (use 70% as a threshold) and save the text data, together with its positions in the original image into the data file.



Future Improvements

Here is a list of potential improvements on preprocessor:

- Explore more possibility of multi-threading to improve parallelization.
- Refine merging algorithms (possibly using hash tables) to make line merging and vertex merging more efficient.
- Detect tilted lines (possibly by refining Hough Transform).
- Improve text detection rate by ‘smoothing’ the image after line removal.
- Rewrite the code using C++ (together with OpenMP, OpenCV2).
- Use Json module to write data into Json file.

Special Thanks

NavPal Floor Plan Group, Hatem Alismail, Maxime Bury,

Professor David Kosbie, Professor Bernardine Dias