Wiki

Overview

Introduction

This wiki page aims to provide an overview of the architecture of our analysis method, briefly describing the main steps that cooperate to build the regression models for SLAM performance prediction.

Description

The analysis consists of four steps:

1. Computation of the localization error performance metric (single run)
2. Computation of the localization error performance metric (environment)
3. Extraction of the geometrical, topological and structural features of the environments
4. Model learning

Computation of the localization error performance metric (single run)

The goal of this phase is to compute a set of measures that summarise the performance of a given SLAM algorithm on each dataset, as described by the Freiburg metric evaluator.
For each dataset in the collection, several stage simulation runs are performed. Each of them is then evaluated by the metric evaluator, computing both translational and rotational errors, in the following way:

1. A first evaluation is performed on a reduced set of randomly extracted relations to assess the error variance;
2. A new evaluation is performed on a more comprehensive set that contains a number of randomly extracted relations that is proportional to the estimated error variance.

Both these operations are implemented and managed by the generateAll.py python script.

As a result of these two operations, we obtain the following performance statistics: **mean, standard deviation, min, max** for both the **translational** and **rotational** components of the localization error of the individual run. The **number of sampled relations** is also recorded.

For the purposes of our analysis, we only consider the mean of the translational and rotational errors as characterizing measures of an individual run; however, the standard deviation and the number of used samples are also available.

Computation of the localization error performance metric (environment)

The goal of this phase is to compute the mean and standard deviation of the translational and rotational components of the localization error of an *environment*, by analyzing the performances obtained by the SLAM algorithm on the individual runs of the environment.

The result of this phase is a series of statistics that concisely describe the average performance of the given SLAM algorithm on each environment and that are denoted as following:

- **transError.mean.mean** is the average of the means of the translational error of all the runs
- **transError.mean.std** is the standard deviation of the means of the translational error of all the runs
- **transError.std.mean** is the average of the standard deviations of the translational error of all the runs
- **transError.std.std** is the standard deviation of the standard deviations of the translational error of all the runs
- **transError.numSamples.mean** is the average of the number of sampled relations for the translational error of all the runs
- **transError.numSamples.std** is the standard deviation of the number of sampled relations for the translational error of all the runs
- **rotError.mean.mean** is the average of the means of the rotational error of all the runs
- **rotError.mean.std** is the standard deviation of the means of the rotational error of all the runs
- **rotError.std.mean** is the average of the standard deviations of the rotational error of all the runs
- **rotError.std.std** is the standard deviation of the standard deviations of the rotational error of all the runs
- **rotError.numSamples.mean** is the average of the number of sampled relations for the rotational error of all the runs
- **rotError.numSamples.std** is the standard deviation of the number of sampled relations for the rotational error of all the runs

Feature extraction

The third phase of our method is the extraction of characterizing features of the environments.
For our analysis, we consider features belonging to three different categories: geometrical, topological, and structural.

Geometrical features

In order to compute useful geometrical features of environments, our method relies on the layout reconstruction methodology implemented by M. Calabrese and V. Arcerito to identify the layout (i.e., the disposition of doors, walls and rooms) of an indoor environment. This tool is included in our distribution with some minor tweaks, and has to be invoked manually via the `batch_extract.py` script in the predictor directory. The parameters of the tool, including the path of the input directory with the ground truth floor plan images to be analyzed and the path of the output directory where to save the extracted layouts, are specified in the `parameters.py` python script.

For each ground truth floor plan image, a directory is created with several files in it, of which the most important for our analysis is the XML file containing all the information about the reconstructed layout; the remaining files are only useful for diagnostic purposes.

From this XML file, our tool extracts a variety of metrics, both at a **local** and at a **global** level.
From a local perspective, we collect the **perimeter, area, longestWall/shortestWall ratio of each room** - both for the room polygon and for its approximating bounding box (potentially useful for non-rectangular rooms).
At a global level, we collect the **perimeter, area, longestWall/shortestWall ratio and the number of rooms of the whole floor**.

Topological features

In addition to the geometrical features extracted in the previous step, we also exploit the same XML files obtained with the layout reconstruction methodology of M. Calabrese and V. Arcerito to obtain information about the topology of each environment.
To do so, we build a topological graph of the environment, where nodes correspond to rooms and an edge between any two nodes represents the existence of a door or passage between the two corresponding rooms.
The algorithm to build the topological graph receives in input the list of identified rooms (as produced by the layout reconstruction method) and the ground truth image of each environment. Then, it executes the following two steps:

- First, it identifies bordering rooms, i.e., rooms that share a wall. This is accomplished by creating a rectangle for each wall of each room that is slightly wider than the true width of the wall, and subsequently looking for intersections between walls belonging to different rooms.
- Second, it classifies bordering rooms as either connected rooms, i.e., rooms that are directly reachable from one another through a door or a passage, and plain bordering rooms, i.e., rooms that just happen to be neighbors but are not connected to each other. This is accomplished by creating a masked version of the ground truth image in which only the area associated with the two rooms is left visible, while everything else is set to black, and performing a floodfill from the representative point of one of the two rooms. If the representative point of the other room gets colored, then the two rooms are connected.

Once the topological graph has been identified, we are able to extract several features, including (but not limited to) the graph's radius, diameter, average shortest path length, number of nodes, number of edges, density, average and standard deviation of betweenness/closeness/eigenvector/katz centrality.

Structural features

Finally, we also build for each environment a simplified, skeletonized representation of its structure in the form of a Voronoi graph. The graph is built starting from a ground truth image of the environment according to the algorithm implemented by Bormann et al. in the ipa_room_segmentation ROS package. Once the graph is visually extracted as an image, we then apply several post processing measures to shrink it down to an actual usable graph:

- first, we use OpenCV to perform a dilation operation on the bitmap Voronoi graph
- then, we run the skeletonization method provided by the skimage python module on the dilated image to build a version of the bitmap Voronoi graph where each line is exactly one pixel wide with no imperfections
- after that, we consider each pixel as a node and we add an edge between two nodes if the two corresponding pixels are close to each other (meaning that one sits within the 3x3 pixel area centered around the other)
- finally, we perform a graph sparsification stage by removing all 'pass-through' nodes, i.e., nodes that have exactly two neighbors, replacing them with an edge that connects their neighbors whose weight is the sum of the weights of the original edges it's replacing.

Once the Voronoi graph has been identified, we are able to extract the same set of features that we also extracted for the topological graph: radius, diameter, average shortest path length, number of nodes, number of edges, density, average and standard deviation of betweenness/closeness/eigenvector/katz centrality.

In addition, we can also estimate the total distance and rotation that the robot would have to travel to fully explore the environment while moving exclusively along trajectories belonging to the Voronoi graph. The pseudocode for the computation of these features is included below.

Voronoi exploration algorithm

```
''' Explore the voronoi graph; return the total amount of travelled distance and the sum
of the top N visited nodes.'''
def exploreVoronoiGraph(VG, image, gtImage, start, voronoiNodesMap, voronoiNodesReverseM
    laserLength *= scale
    minRotDistance *= scale
    # set every node as 'not seen' and with zero visits
    initNodeVisitedStatus(VG)
    # initialize counting variables
    height, width = image.shape
    totalNodes = len(VG.nodes())
    currentPixel = start
    currentAnglePixel = currentPixel
    previousPixel = (currentPixel[0]+1,currentPixel[1])
    numSeenNodes, partialDistance, totalDistance, partialAngle, totalAngle, robotAngle,
    # line-of-sight: identify and mark as 'seen' the pixels that are visible from the cu
    visibleNodes, nseen = lineOfSight(VG, image, gtImage, totalNodes, currentPixel, prev
    numSeenNodes += nseen
    step+=1
    # we proceed until we've seen every single node of the graph
    while numSeenNodes < totalNodes:
        # and until then, we keep looking for the next frontier (shortest euclidean dist
        nearestPixel = retrieveNearestFrontier(image, currentPixel)
        nearestNode = voronoiNodesMap[height-nearestPixel[0], nearestPixel[1]]
        currentNode = voronoiNodesMap[height-currentPixel[0], currentPixel[1]]
        # then, we follow the shortest path that connects our current location to the id
        shortest_path = nx.shortest_path(VG, currentNode, nearestNode)
        for n in shortest_path:
            # each node of the path gets visited and its neighboring nodes in line-of-si
            newPixel = (height-voronoiNodesReverseMap[n][0],voronoiNodesReverseMap[n][1]
            newNode = n
            # increase the total travelled distance by the amount joining two consecutiv
            travelledDistance = node_distance(VG,currentNode,newNode)
            partialDistance += travelledDistance
            totalDistance += travelledDistance
            if partialDistance > minRotDistance:
                partialAngle = math.atan2(newPixel[0]-currentAnglePixel[0], newPixel[1]-
                totalAngle += math.fabs(math.atan2(math.sin(partialAngle-robotAngle),mat
                robotAngle = partialAngle
                currentAnglePixel = newPixel
                partialDistance = 0
            previousPixel = currentPixel
            previousNode = currentNode
            currentPixel = newPixel
            currentNode = newNode
            visibleNodes, nseen = lineOfSight(VG, image, gtImage, totalNodes,currentPixe
            # add to the number of seen nodes the new ones we have actually seen in line
            numSeenNodes += nseen
            step += 1
            # and increase the number of times this current node has been visited (curre
            VG.node[n]['nvisits'] += 1
        currentPixel = nearestPixel
    return totalDistance/scale, totalAngle

''' Retrieve pixels that are visible in line-of-sight given the current robot location a
then, set them to gray, set the corresponding nodes as seen and return them.'''
def lineOfSight(VG, image, gtImage, totalNodes, currentPixel, previousPixel, voronoiNode
    height, width = image.shape
    visiblePixels = retrieveVisiblePixels(image, gtImage, totalNodes, currentPixel, prev
    visibleNodes = pixelsToNodes(visiblePixels, voronoiNodesMap, height)
    markVisiblePixelsAsVisited(image, visiblePixels)
    nseen = markVisibleNodesAsVisited(VG, visibleNodes)
    return visibleNodes, nseen

''' Return the nearest frontier, in euclidean distance, from the current robot location.
def retrieveNearestFrontier(img, currentPixel):
    black_pixels = np.argwhere(img == 0)
    distances = np.sqrt((black_pixels[:,0] - currentPixel[0]) ** 2 + (black_pixels[:,1]
    nearest_index = np.argmin(distances)
    return black_pixels[nearest_index]

''' Retrieves all pixels that are within a certain range from the current pixel and that
are within the simulated field of view of the laser scanner. '''
def retrieveNearbyPixels(image, currentPixel, previousPixel, laserLength, laserFOV):
    # Retrieve all black pixels (black = occupied)
    black_pixels = np.argwhere(image == 0)
    # Compute the orientation of the robot
    ref_angle = math.atan2(currentPixel[0]-previousPixel[0], currentPixel[1]-previousPix
    # Compute the distance between the current pixel and every other black pixel
    distances = np.sqrt((black_pixels[:,0] - currentPixel[0]) ** 2 + (black_pixels[:,1]
    # Compute the relative orientation
    angles = np.arctan2(black_pixels[:,0] - currentPixel[0], black_pixels[:,1]-currentPi
    lgt = len(distances)
    # Return only those pixels that are within the laser range and field of view
    indices = [i for i in range(0, lgt) if distances[i] < laserLength and math.fabs(math
    return black_pixels[indices]

''' Retrieves all pixels that are visible from the current robot location and orientatio
A pixel is considered to be visible if three conditions hold:
- it is within the laser range
- it is within the laser field of view
```

```
  - there are no other black pixels on the line connecting it to the current robot locatio
def retrieveVisiblePixels(image, gtImage, totalNodes, currentPixel, previousPixel, laser
    height, width = image.shape
    # Retrieve candidate pixels
    nearbyPixels = retrieveNearbyPixels(image, currentPixel, previousPixel, laserLength,
    visiblePixels = []
    # If two black pixels are visible from each other, it means no other obstacle (black
    # between them
    for p in nearbyPixels:
        pixelsAlongLine = createLineIterator([currentPixel[1],currentPixel[0]],[p[1],p[0
        black_pixels = np.argwhere(pixelsAlongLine == 0)
        if len(black_pixels)==0:
            visiblePixels.append(p)
    return visiblePixels
```

Model learning

For each of the extracted features, the tool builds a linear regression model of their relationship with each of the previously mentioned performance metrics. For local features, an average over all the rooms is performed.
In addition, the tool also verifies the usefulness of more complex, multiple features regression models.

Each model is then evaluated in terms of degree of correlation (*r-squared coefficient*) and average prediction accuracy (*root mean square error*, or *RMSE*) on test data.

Updated 2 days ago