Описание алгоритма навигации «Proximity»

Алгоритм навигации «Proximity» предназначен для работы в случаях, когда помещение представляет собой длинный и узкий коридор, переходы между этажами по небольшим лестницам и т.д. В этом случае установка маяков для использования метода «Трилатерация» не оправдано, так как потребует значительного количества устройств. Алгоритм «Proximity» позволяет обеспечить необходимую точность позиционирования пользователя при относительно малом количестве маяков.

На рис. 1 показан пример карты помещения, где целесообразно использование алгоритма «Proximity». Структурная схема всего алгоритма приведена на рис. 2.

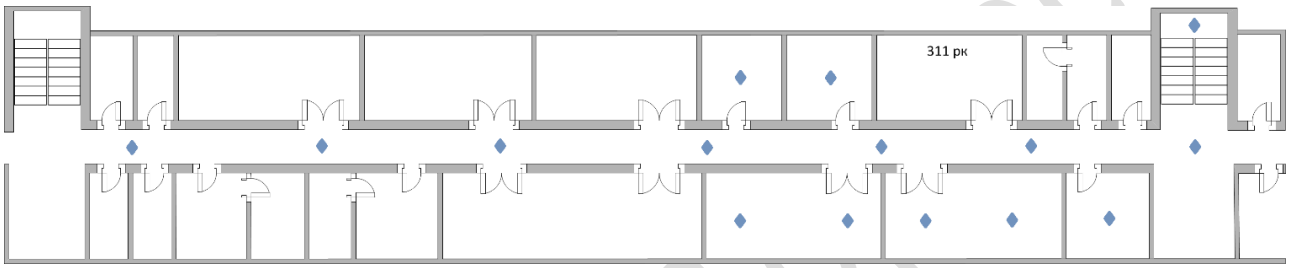

Рис. 1 – Карта и пример расположения маяков при использовании алгоритма «Proximity» (синими ромбами показаны места расположения маяков)

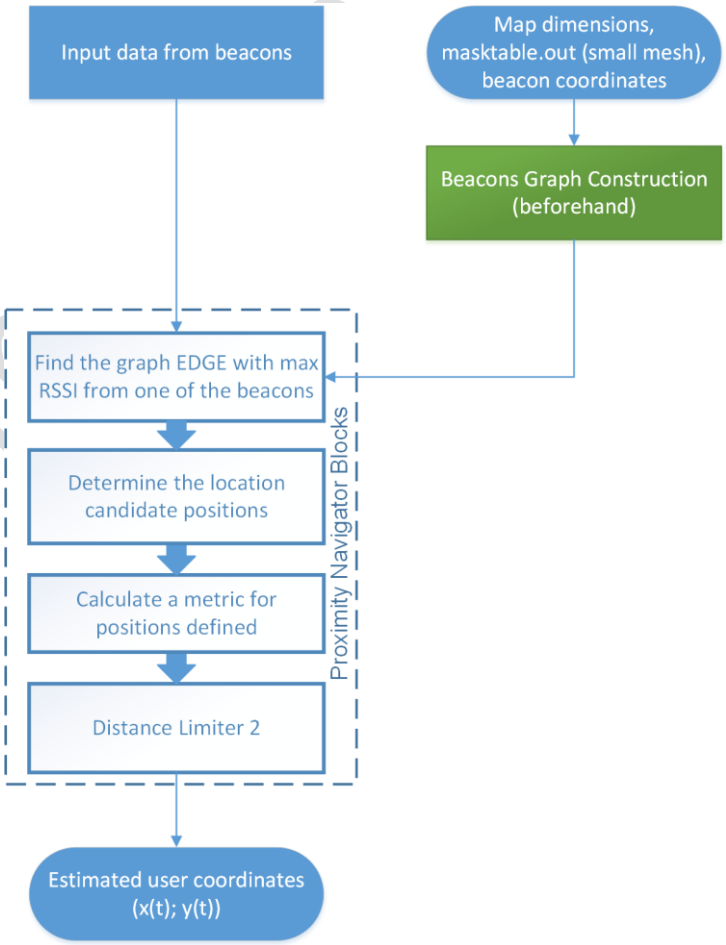

Рис. 2 – Структурная схема алгоритма «Proximity»

Часть 1 – Описание блока построения графа

Первым функциональным блоком в алгоритме «Proximity» является блок Построения графа с использованием положений маячков (на рис. 2 выделен зеленым цветом). Для работы данному блоку требуются:

- физические размеры карты;
- координаты маяков, а также значения их параметров «*damp*» и «*TXpower*»;
- файл мелкого мэша «*masktable*01.*out*» (либо не 01, а какой-либо другой размер).

Построенный граф состоит из узлов в месте установки маяков и ребер, соединяющих эти узлы. При построении ребер используется логика, согласно которой узлы графа не могут быть соединены ребром, которое пересекает стену либо любую другую преграду.

Алгоритм расчета графа (его блок-схема показана на рис. 3 или по ссылке - https://www.dropbox.com/s/ziv1uxjs3cylr6e/Proximity%20%231.1.png?dl=0) выполняется один раз до начала навигации, как только становится известной информация об этаже и, следовательно, о маяках. Алгоритм состоит из двух этапов:

1. Поиск всех возможных ребер между вершинами (маяками) графа, которые не пересекают препятствия;
2. Выделение из набора найденных ребер уникальных (которые формируют уникальный путь из одной вершины в другие).

Рассмотрим каждый этап по отдельности.

Описание блок-схемы алгоритма Этапа 1

Входные параметры:

beacons – массив, каждый элемент которого содержит информацию о физическом маяке, а именно его координатах X и Y (beacons[i].x и beacons[i].y для i-го маяка);

mesh2Rows – количество строк в мелкой сетке карты;

maskTable2 – вектор-столбец, соответствующий мелкой сетке;

mesh2StepX – шаг мелкой сетки по столбцам;

mesh2StepY – шаг мелкой сетки по строкам;

Выходные параметры:

edges – массив ребер графа и их параметров; каждый элемент массива (например, edges[j]=edge) имеет следующие поля:

beac1N (т.е. edge. beac1N) - номер первого из маяков, которые формируют ребро графа, с координатами [edge.beac1X; edge.beac1Y];

beac2N (т.е. edge. beac2N) - номер второго из маяков, которые формируют ребро графа, с координатами [edge.beac2X; edge.beac2Y];

dist (т.е. edge. dist) – длина ребра графа;

angle (т.е. edge. angle) – угол наклона ребра к оси X;

edgeN – номер ребра (важный параметр в рамках последующих алгоритмов).

Алгоритм состоит из двух циклов: 1) по всем маякам как вершинам графа (переменная *i*); 2) по всем маякам кроме *i*-го (переменная *j*) для проверки возможности формирования ребра между *i*-м и *j*-м маяками.

Если прямая, соединяющая два маячка не проходит через черный пиксель (создание ребра возможно), то функция «*isWallBeingCrossed*» (название функции в MatLab) возвращает false. Если ребро может быть создано, то формируется объект «*edge*» и в его поля записываются соответствующие значения (доступные поля описаны выше). Отмечу, что для определения поля «*angle*» используется функция «*calcEdgeAngle*(…)». Далее текущий объект «*edge*» записывается в массив «*edges*» (индекс данного массива «*m*»).

По причине того, что по умолчанию (без дополнительной обработки) маяки, находящиеся в одном коридоре, могут быть развешены в произвольном порядке (не на одной прямой в плоскости XY0), то в результате работы данного алгоритма часто создаются ребра, которые слабо различаются между собой и дублируют более простые соединения маяков (так называемые «ложные» ребра) (см. рис. 4).

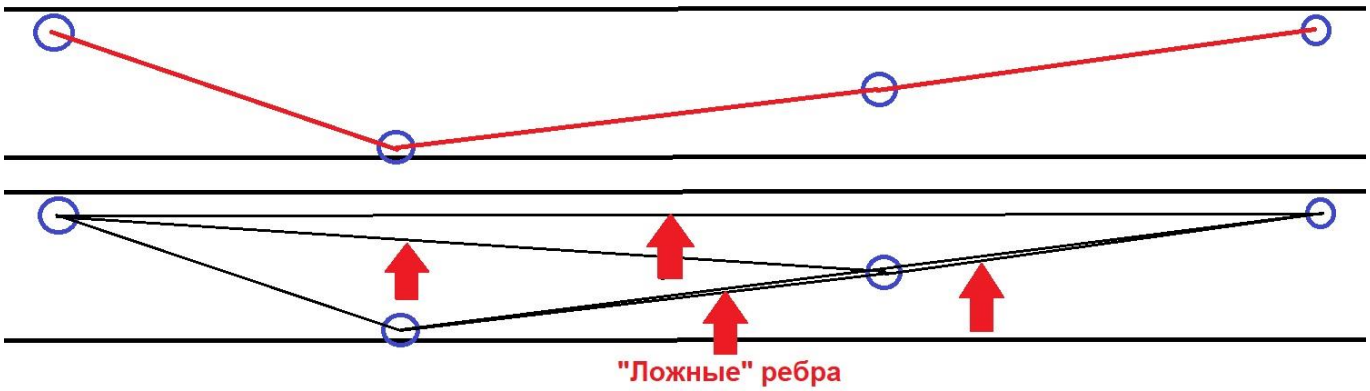

Рис. 4 - Корректный граф (вверху) и исходный граф с «ложными» ребрами (внизу)

Для того, чтобы убрать избыточность в алгоритм добавлена функция «*findCorrectEdges*()», которая отсеивает ложные ребра, созданные для текущего *i*-го маяка (алгоритм функции описан ниже). Ребра, которые признаны корректными, записываются в массив «*correctEdges*» (текущее количество ребер, записанных в данный массив, определяется значением «*eN*»), а содержимое массива «*edges*» очищается.

На выходе всего алгоритма имеем граф, представленный в виде массива объектов типа «*edge*» («*correctEdges*»).

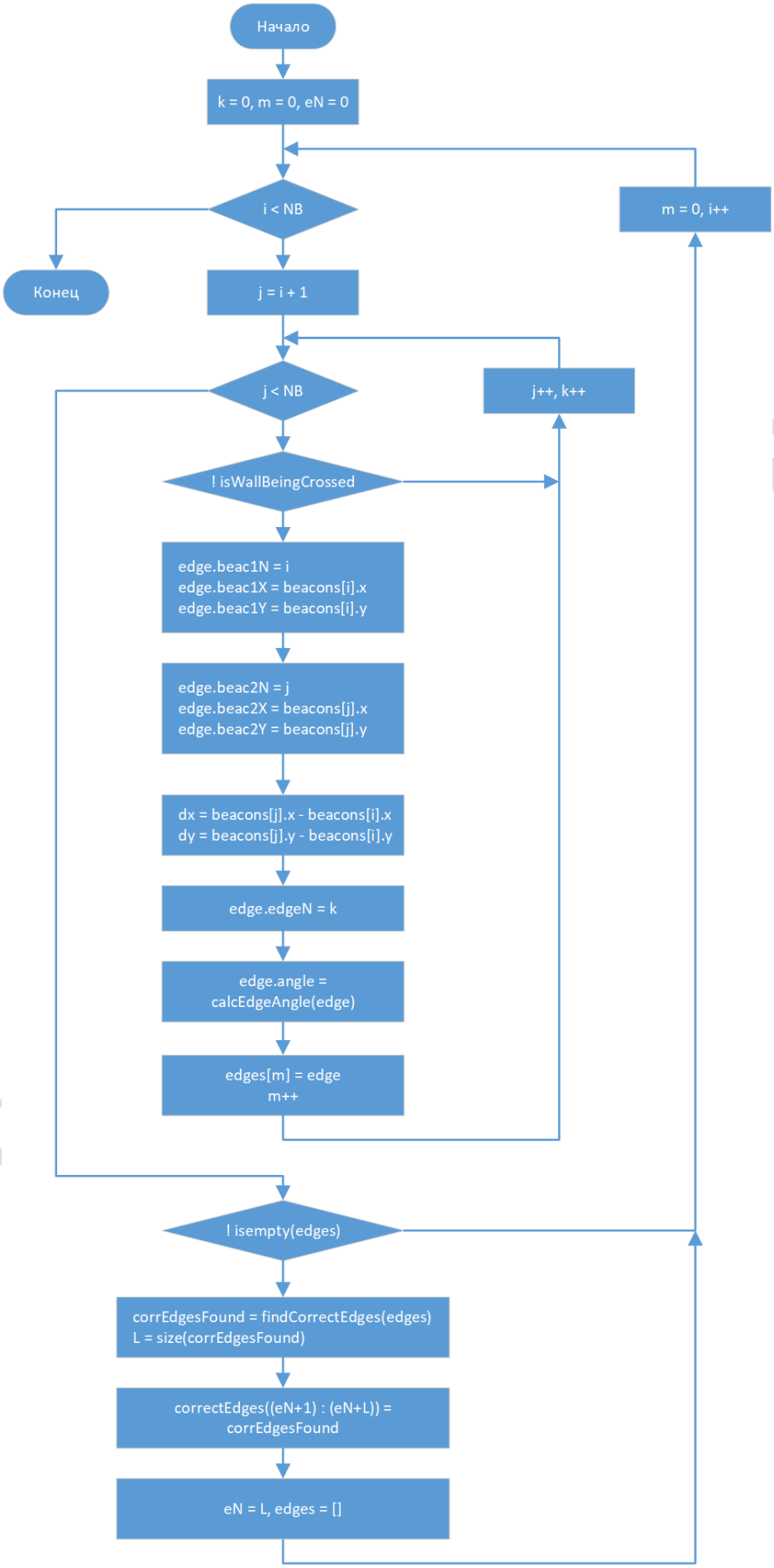

Рис. 3 – Блок-схема основной части алгоритма

Описание блок-схемы алгоритма Этапа 2 (функция «*findCorrectEdges*()»)

Входным параметром функции является массив ребер «*edges*», сформированный основным алгоритмом на Этапе 1. Такой массив формируется для каждого маяка на этаже.

Алгоритм, реализуемый данной функцией, последовательно проверяет каждый ребро из «*edges*», сравнивая его с остальными (т.е. с теми, *edge.dist* которых не равна -1) на соответствие двум условиям:

1. Разность между углом наклона *i*-го ребра к *j*-му не должна превышать 15º (настраиваемый параметр SDK);
2. Если условие 1 выполняется, то из всех ребер, для которых оно выполняется выбирается ребро с минимальной длиной.

Перебор ведется до тех пор, пока не останется либо одно ребро, либо несколько, углы между которыми будут больше 15º.