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. 1.1 показан пример карты помещения, где целесообразно использование алгоритма «Proximity». Структурная схема всего алгоритма приведена на рис. 1.2.

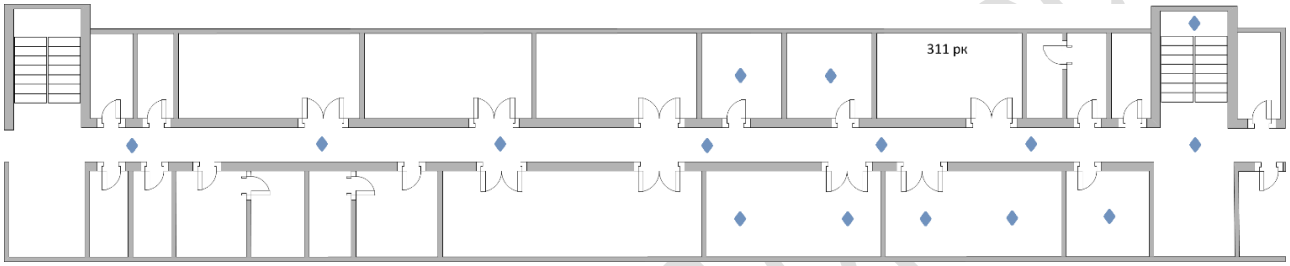

Рис. 1.1 – Карта и пример расположения маяков при использовании алгоритма «Proximity» (синими ромбами показаны места расположения маяков)

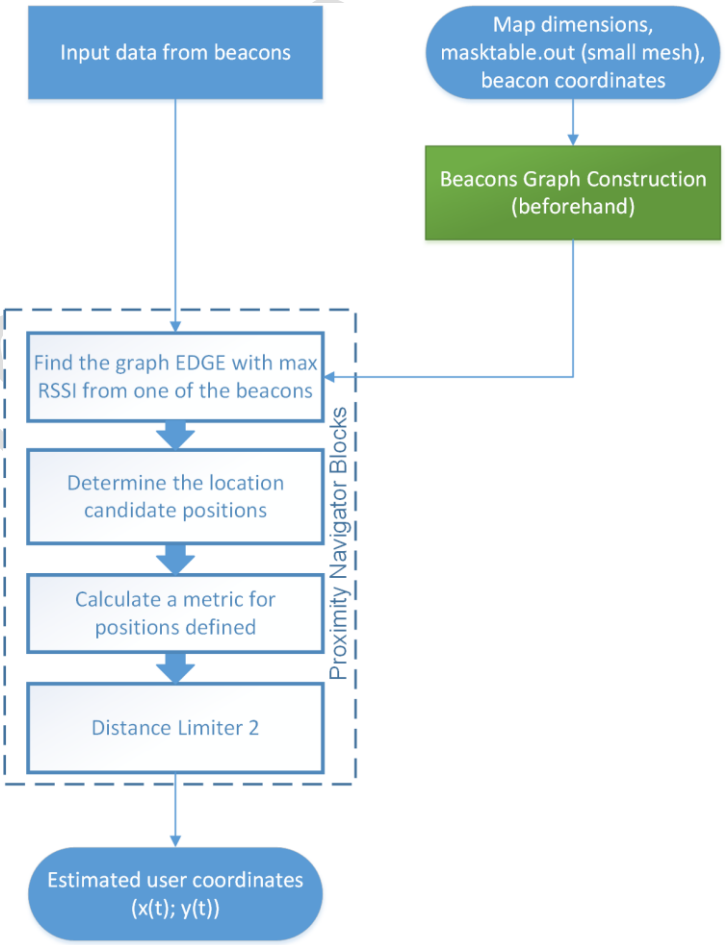

Рис. 1.2 – Структурная схема алгоритма «Proximity»

Часть 1 – Описание блока построения графа

Первым функциональным блоком в алгоритме «Proximity» является блок Построения графа с использованием положений маячков (на рис. 1.2 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. 1.3 или по ссылке - https://www.dropbox.com/s/ciw75i6p68tvn8e/Proximity%20%231.2.png?dl=0) выполняется один раз до начала навигации, как только становится известной информация об этаже и, следовательно, о маяках. Условно состоит из 2 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 блок-схему алгоритма (рис. 1.3).

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:

correctEdges – массив корректных ребер графа и их параметров;

edge - элемент массива «edges» (например, edges[j]), 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 для последующих алгоритмов).

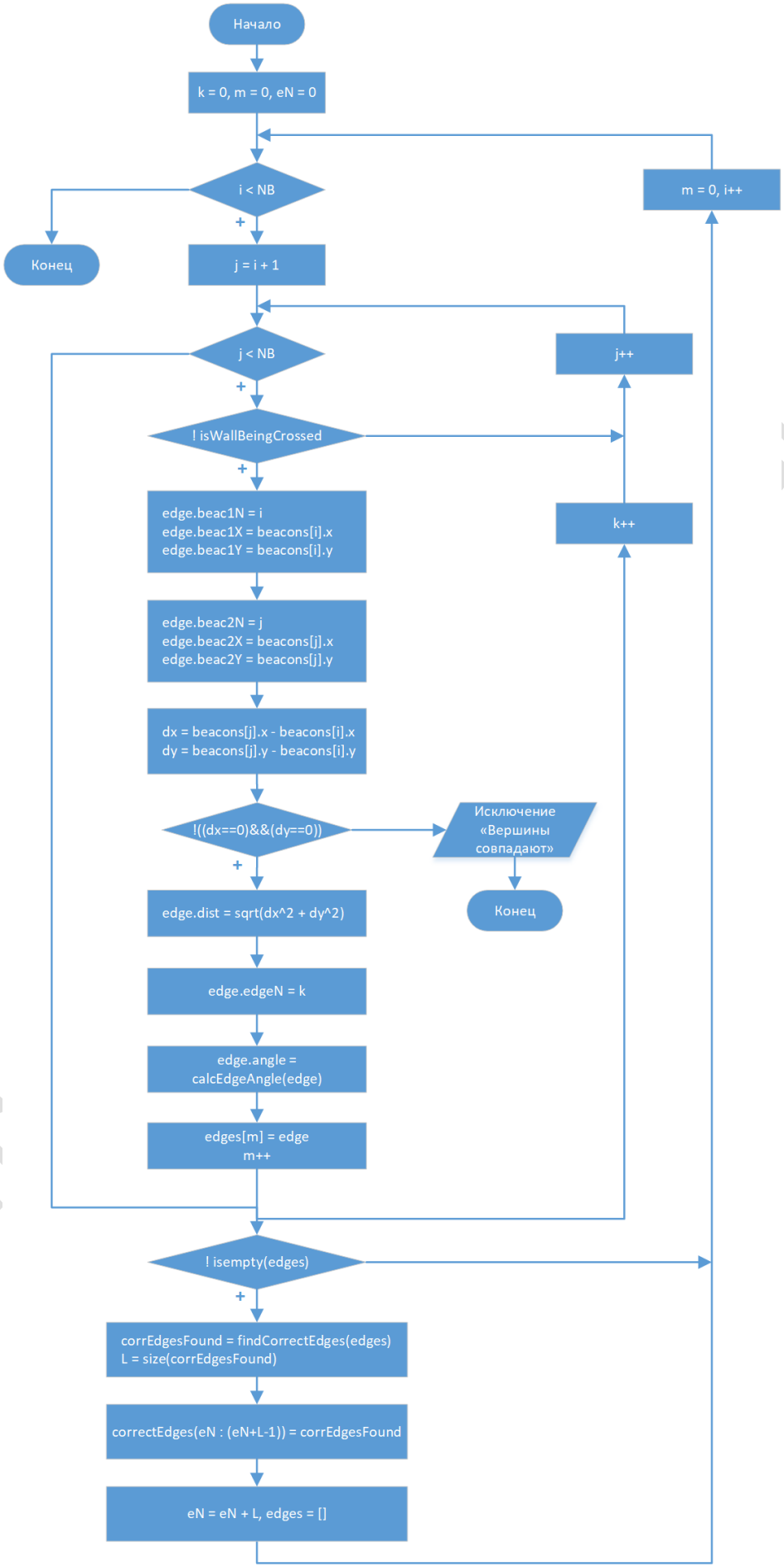

Рис. 1.3 – Блок-схема основной части алгоритма

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

Если прямая, соединяющая два маячка не проходит через черный пиксель (функция «*isWallBeingCrossed*» (название функции в MatLab) возвращает false), то ребро может быть создано. В этом случае формируется объект «*edge*» и в его поля записываются соответствующие значения. 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, например, 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 (имеют близкие значения углов) и дублируют более простые соединения маяков (так называемые «ложные» ребра) (см. рис. 1.4).

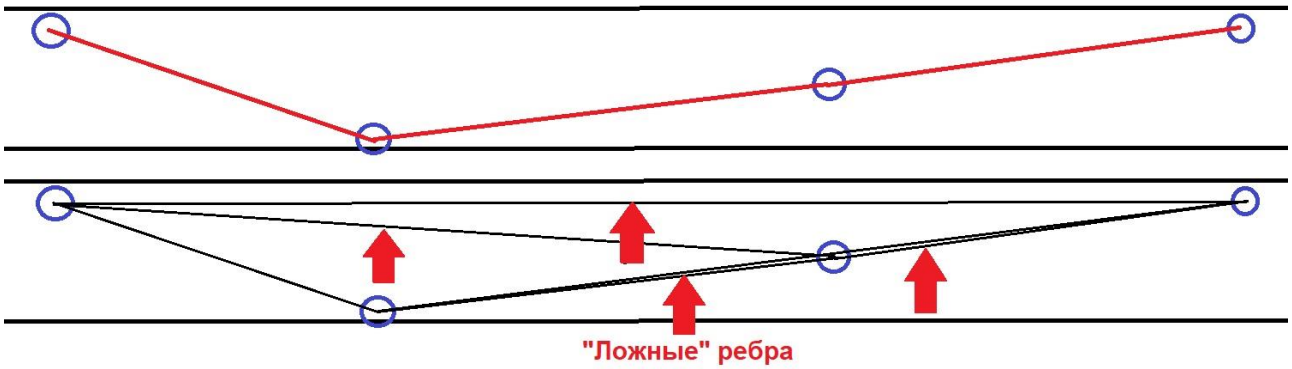

Рис. 1.4 - Корректный граф (вверху) и
исходный граф с «ложными» ребрами (внизу)

Для того, чтобы убрать избыточность в алгоритме существует функция «*findCorrectEdges*()», которая находит и отсеивает ложные ребра (алгоритм функции описан ниже). Ребра, которые признаны корректными, записываются в массив «*correctEdges*» (текущее количество ребер, записанных в данный массив, определяется переменной «*eN*»), а содержимое массива «*edges*» очищается и процесс повторяется снова для следующего маяка.

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

Блок-схема алгоритма определения угла ребра
(функция «*calcEdgeAngle*()», рис. 1.5)

В функцию передается сущность «*edge*» (ребро). Рассчитываются расстояния между вершинами ребра по каждой координате (переменные «*dx*», «*dy*»). Исключительная ситуация, когда вершины совпадают друг с другом обрабатывается в основном алгоритме путем выбрасывания исключения (см. рис. 1.3). Угол вычисляется через арксинус и его значения находятся в диапазоне $[-180^0; 180^0]$.

Из нюансов отмечу преобразование значения угла в градусы и домножение на «-1» с целью перевода угла в систему координат, где ось Y направлена вниз (так как с ней работаем в UA и Indoor Tool). Функция арксинуса, как известно, не позволяет по значению угла определить, где находится точка: в первой или второй четвертях (когда угол положительный), либо в третьей или четвертой четвертях (если значение угла отрицательное). Другими словами, значения угла на выходе функции арксинуса лежат в диапазоне $[-90^0; 90^0]$. Потому в коде эти ситуации отслеживаются и значения угла соответствующим образом преобразовываются в диапазон $[-180^0; 180^0]$.

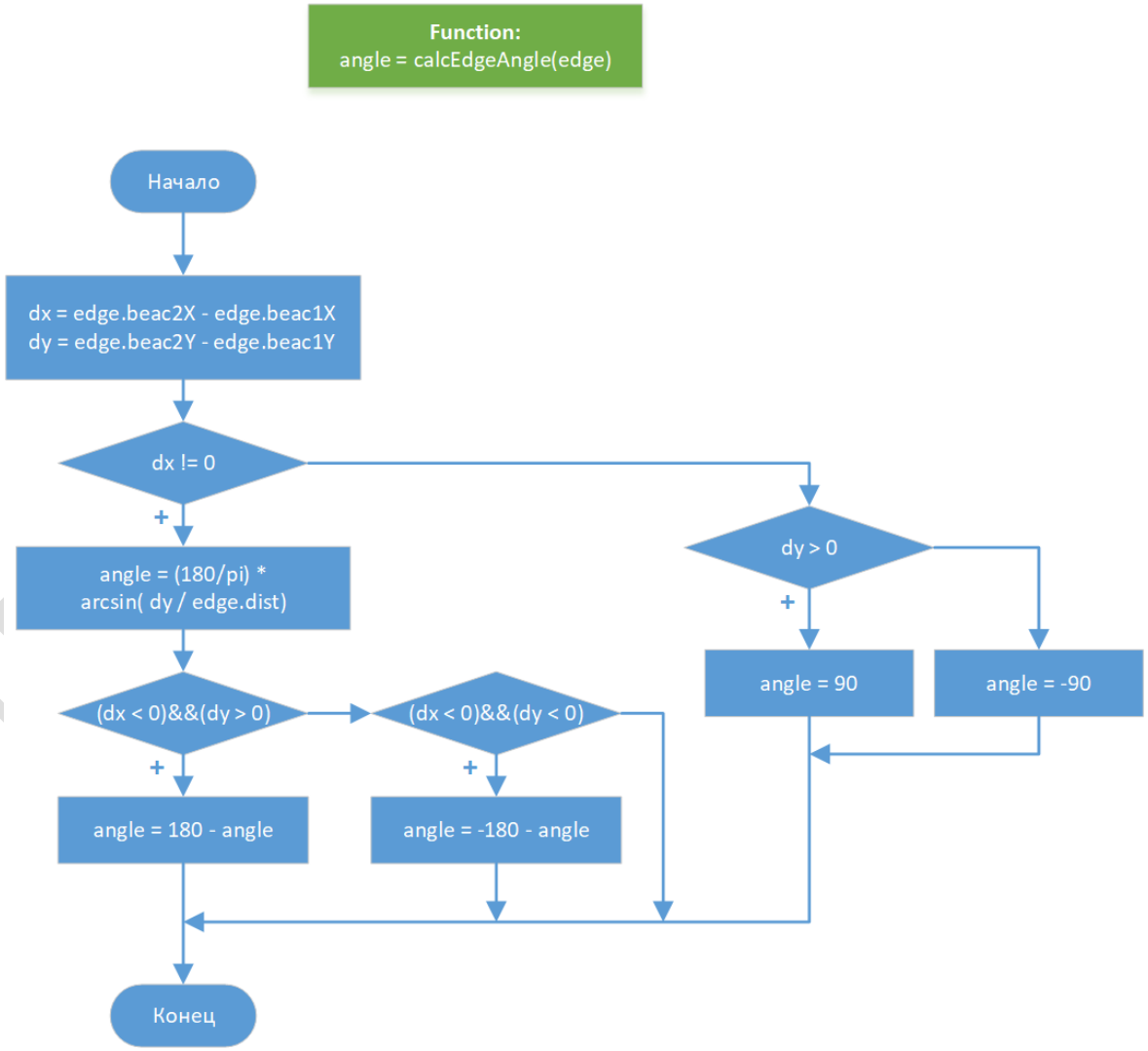

Рис. 1.5 – Блок-схема алгоритма функции «*calcEdgeAngle*(*edge*)»

Блок-схема алгоритма отсеивания «ложных» ребер (функция «*findCorrectEdges*()», рис. 1.6)

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

Алгоритм, реализуемый данной функцией, последовательно проверяет каждое ребро из «*edges*», сравнивая его с остальными ребрами. «Ложным» считается то ребро, значение *edge.dist* которого равно «-1». Такое значение может быть записано в поле ребра при выполнении двух условий:

1. Разность между углом наклона текущего (*i*-го) ребра к одному из ребер из массива «edges» (*j*-му) не превышала 15º, где значение «15º» является одним из параметров алгоритма;

2. При выполнении условия 1 для текущей пары ребер данное ребро имело большую длину (значение поля «*edge.dist*»).

Перебор ведется до тех пор, пока во входном массиве «*edges*» не останется либо одно ребро, либо несколько, углы между которыми будут больше 15º.

Из особенностей отмечу процедуру преобразования углов ребер перед расчетом их разности. Для корректного расчета знаки углов должны быть одинаковы. По этой причине при сравнении значения обеих углов приводятся либо к диапазону [0; 360^0], либо [-360^0; 0].

В заключительной части алгоритма происходит перезапись всех корректных ребер в выходной массив.

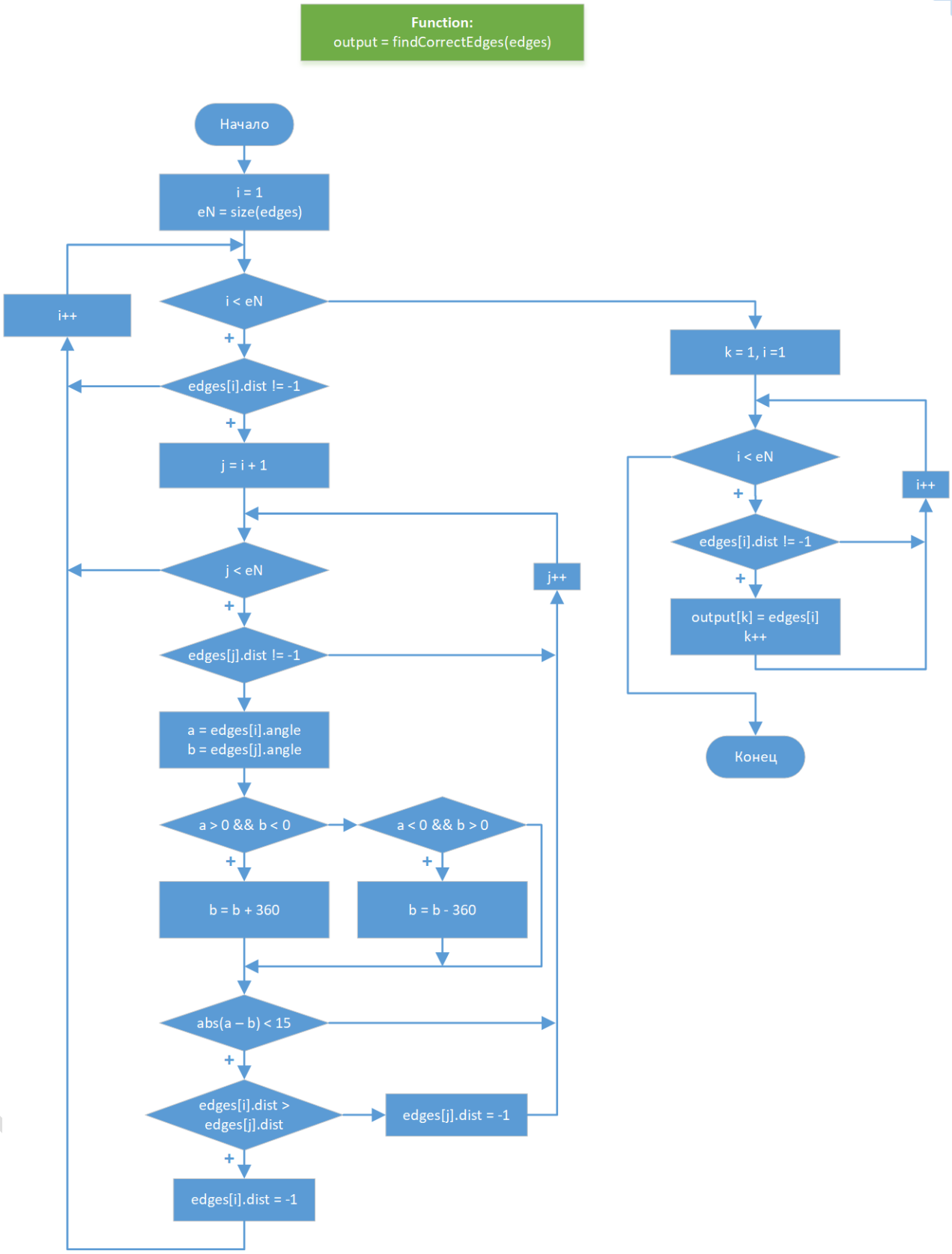

Рис. 1.6 – Блок-схема алгоритма функции «*findCorrectEdges*(…)»

Часть 2 – Описание блока поиска текущего рабочего ребра

На рис. 2.1 показана структурная схема всего алгоритма «Proximity» с указанием блока, который описывается в данном разделе.

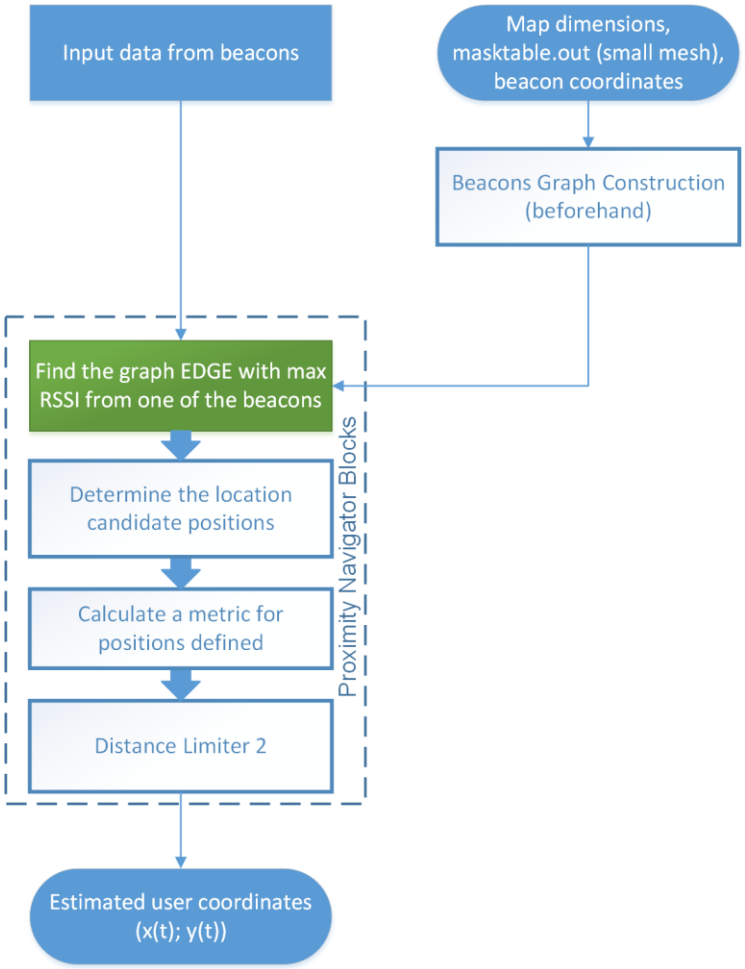

Рис. 2.1 – Структурная схема алгоритма «Proximity»

Блок осуществляет поиск ребра графа, где находится пользователь. Идея данного этапа (и в целом данного алгоритма навигации) состоит в том, что в узких помещениях сигнал от маяка, к которому пользователь находится ближе всего, будет наиболее мощным. Далее находится второй маяк (с наиболее сильным сигналом) на ребрах, которые выходят из найденной вершины.

Формально алгоритм можно разделить на следующие этапы:

1. На первом шаге алгоритм осуществляет сортировку текущих значений RSSI от всех маяков;
2. Из отсортированного массива отбирается маяк (вершина графа) с максимальным RSSI, его номер записывается в переменную «*maxBeacN*»;
3. Далее алгоритм осуществляет поиск всех ребер графа, одна из вершин которых является «*maxBeacN*»-маяком; найденные ребра записываются в массив ребер-кандидатов «*edgeCandidates*»;
4. Из массива ребер-кандидатов «*edgeCandidates*» находится ребро, в котором вторая вершина (маяк) имеет наибольшее по величине значение

RSSI (наибольшее, не включая в рассмотрение значение RSSI от вершины maxBeacN).

В виду того, что операции, перечисленные в этапах алгоритма, довольно простые, не вижу смысла рисовать для них блок-схемы. При необходимости всегда можем обсудить реализацию. Опишу только входные и выходные параметры алгоритма, так как они могут встречаться дальше.

Входные параметры алгоритма:

beaconsRSSI – массив, каждый элемент которого содержит информацию о текущем значении RSSI для *i*-го маяка;

edges – массив ребер графа и их параметров;

edge - элемент массива «*edges*» (например, *edges*[*j*]), имеет следующие поля:

*beac*1*N* (т.е. *edge*. *beac*1*N*) - номер первого из маяков, которые формируют ребро графа, с координатами [*edge*.*beac*1*X*; *edge*.*beac*1*Y*];

*beac*2N (т.е. *edge*. *beac*2*N*) - номер второго из маяков, которые формируют ребро графа, с координатами [*edge*.*beac*2*X*; *edge*.*beac*2*Y*];

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

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

edgeN – номер ребра (важный параметр для последующих алгоритмов).

Выходные параметры алгоритма:

foundEdge – один из элементов массива «*edges*», который выбран алгоритмом.

Часть 3 – Описание блока поиска положений-кандидатов

На рис. 3.1 показана структурная схема всего алгоритма «Proximity» с указанием блока, который описывается в данном разделе.

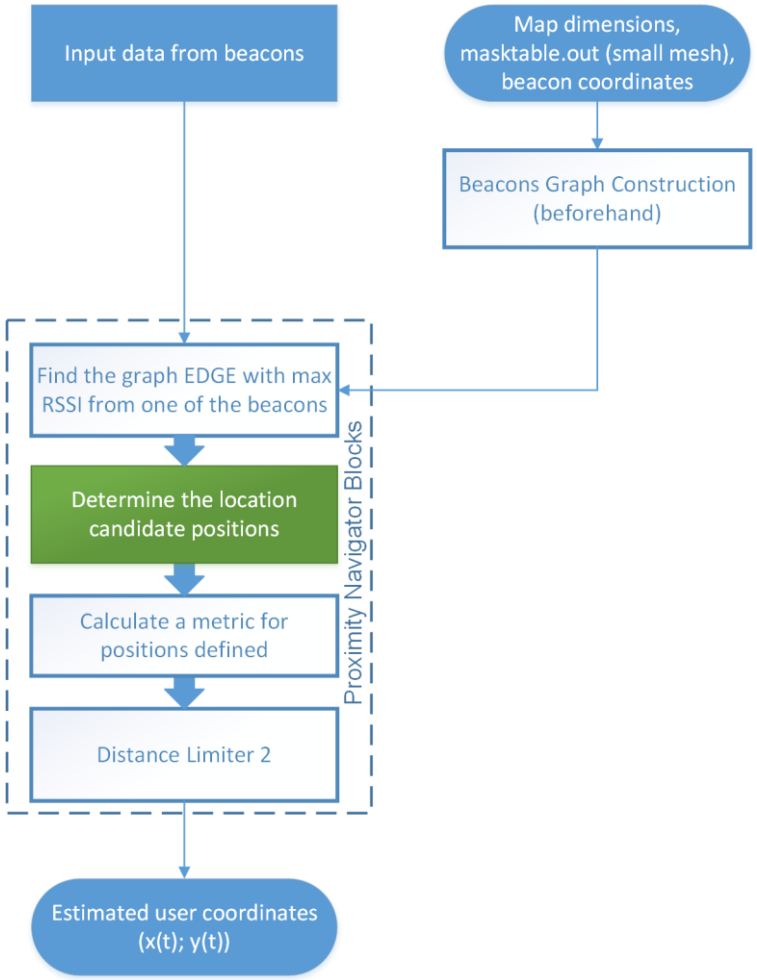

Рис. 3.1 – Структурная схема алгоритма «Proximity»

Блок осуществляет поиск всех возможных положений пользователя на ребре, выбранном в предыдущем блоке (см. часть 2). Физический смысл алгоритма состоит в том, что RSSI от каждого маяка можно пересчитать в расстояние от вершины графа (маяка) до положения пользователя. Данное расстояние физически является радиусом окружности с центром в определенной вершине. В идеальных условиях такие окружности должны иметь одну общую точку пересечения с выбранным в части 2 ребром в месте текущего положения пользователя. Однако на практике наиболее точные результаты наблюдаются только для окружностей с центрами в ближайших вершинах (в вершинах выбранного ранее ребра). Физически это объясняется тем, что чем ближе пользователь к определенному маяку, тем меньше флуктуирует сигнал от него на выходе приемника, и, кроме того, из-за логарифмического вида path-loss модели распространения сигнала погрешность от неправильно рассчитанного расстояния в ближней зоне меньше, чем в средней и дальней.

Тем не менее, в данном блоке ищутся точки пересечения окружностей от всех активных маяков со любыми ребрами графа. Далее из всех точек выбираются те, которые лежат на ребре, найденном в предыдущем блоке.

Формально алгоритм можно разделить на следующие этапы:

1. Для каждого маяка этажа карты (вершины графа) проверяется наличие RSSI на выходе RSSI Kalman Filter;
2. Если на выходе фильтра пакет для заданного маяка есть, то при помощи модели распространения сигнала (path-loss model), параметров TXpower и damp маяка, текущего значения RSSI происходит расчет расстояния от маяка до предполагаемого положения пользователя;
3. Зная текущий радиус окружности (расстояние до пользователя), находятся все точки пересечения окружности и ребер графа; в алгоритме предусмотрена возможность находить пересечения с произвольным количеством ребер. Поиск точек пересечения (положений-кандидатов) осуществляется функцией «*findCrossings*», которая описана ниже;
4. Для нахождения пересечения/й окружности и ребра/ребер решается квадратное уравнение; возможны ситуации, когда функция «*findCrossings*» вернет один или два корня, либо не вернет корней вообще;
5. Корректные пересечения (положения-кандидаты) записываются в массив *correctPoints*, а все пересечения – в массиве *allPoints*.

Входные параметры алгоритма:

beaconsRSSI – массив, каждый элемент которого содержит информацию о текущем значении RSSI для *i*-го маяка;

beaconParams – массив, каждый элемент которого содержит данные о координатах маяка (*x*, *y*, *z*), его параметрах *TXpower* и *damp*; предполагается, что порядок следования маяков в массиве *beaconsRSSI* и *beaconParams* совпадают;

edges – массив ребер графа и их параметров;

edge - элемент массива «edges» (например, edges[j]), имеет те же поля, что и подобные сущности, описанные для предыдущих блоков.

Выходные параметры алгоритма:

correctPoints – массив положений-кандидатов (корректных пересечений), найденных алгоритмом; каждое положение-кандидат описывается сущностью *point*;

allPoints – массив всех найденных алгоритмом пересечений;

point - элемент массива «*correctPos*» и «*allPos*», имеет три поля:

x – координата X положения;

y – координата Y положения;

beacN – номер маячка, от которого получено данное положение-кандидат;

Алгоритм поиска точек пересечения окружности и ребер графа (функция «*findCrossings*()»)

На рис. 3.2 графически изображен результат работы данного алгоритма. По сути, метод решает систему уравнений: уравнения прямой, описывающей текущее рабочее ребро, и уравнения RSSI окружности от каждого маяка.

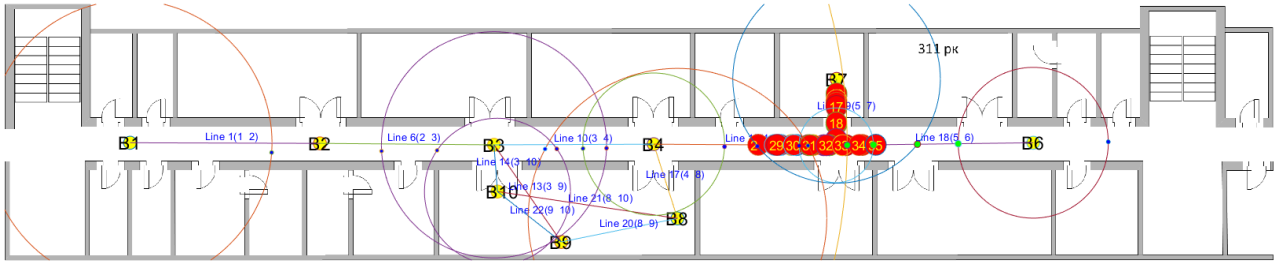

Рис. 3.2 – Графическое представление RSSI окружностей и их точек пересечения с ребрами графа

Входные параметры алгоритма:

distance – расстояние от текущего маяка до пользователя;

currentBeacon = [*xB*, *yB*] – координаты текущего маяка, от которого получен сигнал;

beac1N = [*x*1, *y*1] – координаты первого маяка, формирующего ребро графа;

beac2N = [*x*2, *y*2] – координаты второго маяка, формирующего ребро графа;

Выходные параметры алгоритма:

correctPoints = [*xR*1, *yR*1; *xR*2, *yR*2] – координаты положений-кандидатов, определенные как пересечение между ребром графа и RSSI окружностью (данный массив может быть пустым, либо содержать координаты одной или двух точек);

allPoints – массив всех найденных алгоритмом пересечений; содержит две точки с (0; 0), если дискриминант (Д) уравнения меньше 0; одну точку, если Д равен 0, и две точки, корни уравнения, если Д больше 0.

Формально алгоритм можно разделить на следующие этапы:

1. Проверяется необходимое условие для корректной работы алгоритма – вершина (маяк) A должна быть правее, чем вершина (маяк) B; если это условие не выполняется, то маяки меняются местами (локально);
2. Для текущего активного ребра записывается уравнение прямой;
3. Для заданного маяка записывается уравнение окружности;
4. Далее, решение системы уравнений (уравнение прямой ребра и уравнение окружности) сводится к решению квадратного уравнения, где коэффициенты A, B, C находятся из вершин ребра, радиуса и координат центра окружности:

$$Ax^2 + Bx + C = 0$$

5. Если в результате решения уравнения оказывается, что Д имеет отрицательное значение, то в массив «*allPoints*» записываются две точки, соответствующие началу координат;
6. Если в результате решения уравнения корень/корни есть (Д равен или больше 0), то каждая найденная точка проверяется на принадлежность активному ребру при помощи функции «*isPointCorrect*()»; «ложной» будем называть такую точку, которая находится на продолжении активного ребра, за его пределами;
7. Корректный корень/корни записываются в массив «*correctPoints*», а ложный/ложные в «*allPoints*».

Алгоритм проверки точки пересечения (корня уравнения) (функция «*isPointCorrect*()»)

Функция делает проверку точки пересечения окружностью RSSI активного ребра, чьи координаты получены в результате решения квадратного уравнения.

Входные параметры алгоритма:
point = [*x*, *y*] – координаты проверяемой точки;
beac1N = [*x*1, *y*1] – координаты первого маяка, формирующего ребро графа;
beac2N = [*x*2, *y*2] – координаты второго маяка, формирующего ребро графа;

Выходные параметры алгоритма:
answer – true или false – результат проверки; если точка принадлежит ребру, то выход функции равен true; иначе – false.

В начале проверяется принадлежность координаты *x* точки ребру, т.е. интервалу значений между *beac1N.x* и *beac2N.x*. Если это условие выполнено, то делается проверка координаты *y* точки на принадлежность интервалу значений между *beac1N.y* и *beac2N.y*. Если оба условия выполнены, то *answer* = true, иначе false.

Часть 4 – Описание блока расчета метрики по точкам-кандидатам

На рис. 4.1 показана структурная схема всего алгоритма «Proximity» с указанием блока, который описывается в данном разделе.

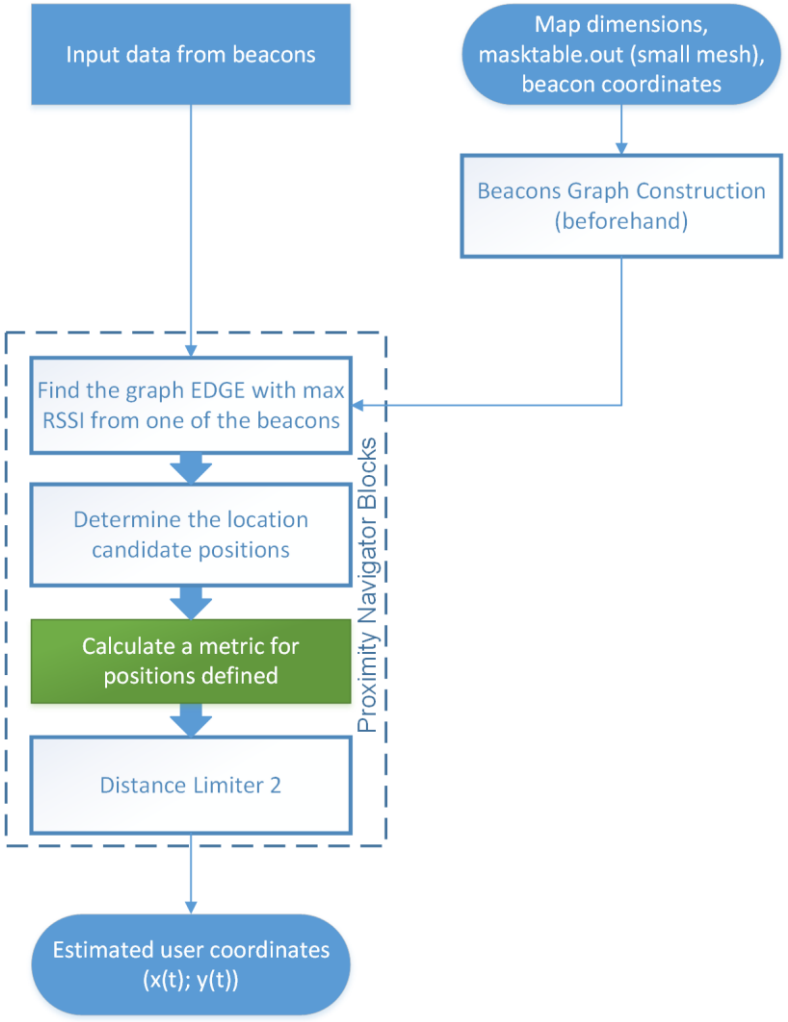

Рис. 4.1 – Структурная схема алгоритма «Proximity»

Блок обрабатывает все найденные корректные (лежащие на ребре) положения-кандидаты. В зависимости от их количества и особенностей определяется результирующее положение пользователя.

Формально алгоритм можно разделить на следующие этапы:

1. Вначале проверяется количество положений-кандидатов, определенных на предыдущем шаге общего алгоритма (часть 3);
2. Если кандидатов нет, метод возвращает *null*; Если кандидат один, то он и является метрикой; в обоих случаях работа блока заканчивается;
3. Если кандидатов более одного, то из всего количества точек определяются те, которые получены от пересечения RSSI окружностей маяков, сформировавших выбранное ребро графа (например, ребро 11-6 сформировано маяками №11 и №6; ищем, какое из положений получено

путем пересечения RSSI окружности от маяка №11 и текущего ребра; также для маяка №6); назовем их «достоверные» положения;

4. Расчет «центрального» значения положения пользователя:
    a. Если в пункте 3 не удалось найти ни одного «достоверного» положения, то все остальные точки положений-кандидатов усредняются и это является результатом работы блока, и он заканчивает на этом свое выполнение;
    b. Если «достоверное» положение одно (второе могло не попасть на текущее активное ребро), то его значение становится «центральным»;
    c. Если найдены оба «достоверных» положения, то «центральное» положение рассчитывается как среднее координат данных двух точек;
5. Если в пункте 4 было определено «центральное» положение, то далее рассчитываются веса каждого положения-кандидата (включая «достоверные» положения) как величина, обратная расстоянию от «центрального» положения до маяка, от RSSI окружности которого эта точка получена;
6. Далее полученные веса нормируются;
7. Результат работы блока рассчитывается как взвешенное среднее от всех положений-кандидатов.

Входные параметры алгоритма:

correctPos – массив положений-кандидатов (корректных пересечений), найденных алгоритмом на этапе 3; каждое положение-кандидат описывается сущностью *point*=[*x y beacN*];

beaconCoords – массив, каждый элемент которого содержит данные о координатах маяка (*x*, *y*, *z*), а также его порядковом номере;

foundEdge – рабочее ребро (сущность «*edge*»), которое выбрано алгоритмом на этапе 2;

Выходные параметры алгоритма:

predUserPos = [*x y*] – предполагаемое вычисленное положение пользователя (которое находится на текущем рабочем ребре).

Часть 5 – Описание блока ограничителя расстояния, на которое может переместиться пользователь

На рис. 5.1 показана структурная схема всего алгоритма «Proximity» с указанием блока, который описывается в данном разделе.

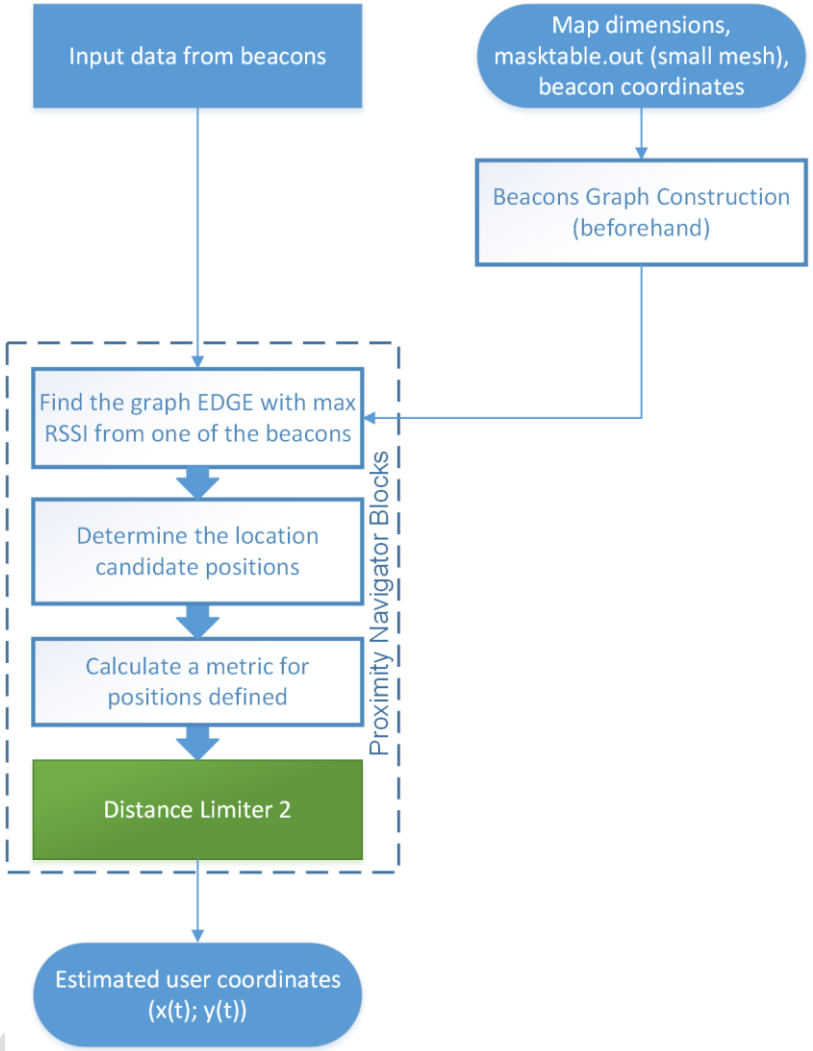

Рис. 5.1 – Структурная схема алгоритма «Proximity»

Блок анализирует предполагаемое положение пользователя, полученное на этапе 4. В случае превышения расстояния между предыдущим положением и предполагаемым текущим, алгоритм корректирует текущее оцененное положение.

Формально алгоритм можно разделить на следующие этапы:

1. Определяется интервал времени между текущим и предыдущим рассчитанными положениями пользователя, а также максимальное расстояние (переменная *maxDist*), которое пользователь может преодолеть за это время (на основании заданной зависимости максимального расстояния от времени);

2. Алгоритм проверяет, находится ли текущее предполагаемое положение (переменная *currentUserPos*) пользователя на том же ребре графа, что и на предыдущем шаге, или нет;
    a. *Вариант 1*: Условие выполняется, т.е. на текущем и предыдущем шагах положения пользователя находятся на одном и том же ребре;
    b. *Вариант 2*: Условие не выполняется, т.е. на текущем и предыдущем шагах положения пользователя находятся на разных ребрах графа;

Ход алгоритма для *Варианта 1*:

3. Проверяется условие, превышает ли расстояние между предыдущим и текущим положениями пользователя допустимую максимальную величину перемещения (для заданного интервала времени между перемещениями); Если не превышает, то алгоритм возвращает предполагаемое положение пользователя *currentUserPos* без изменений и заканчивает свою работу;
4. Если условие в п.3 не выполняется (расстояние превышает максимально допустимую величину), тогда алгоритм вызывает метод *findCrossings*() (см. раздел 3) со следующими параметрами:

(*maxDist*, *prevPos*, *beac1N*, *beac2N*)

где *maxDist* – максимальное расстояние, на которое пользователь может переместиться;

prevPos = [*x*, *y*] – координаты предыдущего положения пользователя;
beac1N = [*x*1, *y*1] – координаты первого маяка, формирующего ребро графа;
beac2N = [*x*2, *y*2] – координаты второго маяка, формирующего ребро графа;

Если функция в качестве результата в переменной *correctPoints* возвращает координаты двух положений, то в качестве ответа выбирается положение, ближайшее к исходно оцененному положению пользователя на текущем шаге решения. Если же возвращаются координаты одной точки, то это положение и является результатом алгоритма.

Ход алгоритма для *Варианта 2*:

5. Общая задача – определить, превышает ли расстояние от предыдущего положения пользователя по ребрам графа до предполагаемого положения на данном шаге величину *maxDist* или нет; для решения данной задачи в начале необходимо получить последовательность ребер (траекторию) между предыдущим и предполагаемым положениями пользователя; для определения такой траектории решили использовать алгоритм поиска кратчайшего пути по графу Дейкстры;
6. Пример использования алгоритма Дейкстры:

Предыдущее положение пользователя находилось на ребре 1–2. Текущее предполагаемое положение пользователя находится на ребре 3–4. Тогда траектория, определенная методом Дейкстры, будет 1–2–3–4.

Последовательность решения задачи в данном случае будет следующей. Вначале алгоритм определяет расстояние от предыдущего положения пользователя до вершины №2 (*dist*1). Уменьшаем величину *maxDist* на *dist*1. Далее проверяем значение (*maxDist* - *dist*1), превышает ли оно длину следующего ребра 2-3.

Если не превышает, то при помощи функции *findCrossings*() находим точку пересечения ребра 2-3 и окружности радиусом (*maxDist* - *dist*1) с центром в вершине 2.

Если значение (*maxDist* - *dist*1) превышает длину ребра 2-3, уменьшаем (*maxDist* - *dist*1) на длину ребра 2-3. Далее повторяем аналогичную проверку со следующим ребром, в данном примере ребром 3-4. И либо находим точку пересечения с ребром, либо вычитаем его длину из величины *maxDist*.

Входные параметры алгоритма:

currentUserPos=[*x y*] – предполагаемое положение пользователя (которое находится на текущем рабочем ребре);

previousUserPos=[*x y*] – предыдущее положение пользователя;

beaconParams – массив, каждый элемент которого содержит данные о координатах маяка (*x*, *y*, *z*), его параметрах *TXpower* и *damp*; предполагается, что порядок следования маяков в массиве *beaconsRSSI* и *beaconParams* совпадают;

edges – массив ребер графа и их параметров;

Выходные параметры алгоритма:

finalPos=[*x y*] – конечное положение пользователя.