СОДЕРЖАНИЕ

1. ВВЕДЕНИЕ

1.1. Наименование проекта

Наименование программы: «Автоматическое планирование траектории», наименование программы на английском: «Automatic path planning».

1.2. Наименование темы

Программа выполнена в рамках курсовой работы на ПМИ по теме «Pathplanning.ru :: автоматическое планирование траектории (2018)».

1.3. Документ, на основе которого ведется разработка

Приказ № 2.3-02/1501-03 от 15.01.2019 «Об утверждении тем и руководителей курсовых работ студентов 2 курса Факультета компьютерных наук, образовательная программа «Прикладная математика и информатика».

2. НАЗНАЧЕНИЕ И ОБЛАСТЬ ПРИМЕНЕНИЯ

2.1. Функциональное назначение программы

Проект предназначен для поиска пути оптимальной длинны в пространстве, заданном графом, с использованием различный алгоритмов и эвристик. Программа является консольным приложением, в качестве аргумента принимающим файл формата .xml, в котором описано пространство, стартовая и конечная точки и ограничения на передвижения объекта. Программа представляет заданное пространство в виде графа и производит поиск оптимального пути. В результате своей работы, приложение создает файл формата .xml, в котором описан оптимальный путь (длинна пути, последовательность вершин, ему принадлежащая), а в случае его отсутствие, сказано о недостижимости конечной точки из начальной.

2.2. Эксплуатационное назначение программы

Приложение может быть использовано в академических целях людьми, занимающимися изучениями алгоритмов поиска кратчайшего пути в пространстве. Программа позволяет сравнивать эффективность разных алгоритмов и эвристик. Другим возможным назначением программы является её применение в практических целях. Программа может быть использована компанией, нуждающейся в нахождение оптимального пути на карте. Примером её использования может являться построение маршрута в городе для робота-доставщика.

3. ТЕХНИЧЕСКИЕ ХАРАКТЕРИСТИКИ

3.1. Постановка задачи на разработку программы

Основная задача, которая была поставлена, - эффективная реализация алгоритмов поиска кратчайшего пути в графах, а именно: реализация алгоритмов: алгоритм Дейкстры, алгоритм A-стар (с использованием эвристики Чебышева, евклидовой, манхэттенской и диагональной эвристик), алгоритм Theta*.

3.2. Алгоритм работы программы

1) Считывание данных из xml файла

2) Запуск алгоритма поиска пути (алгоритм Дейкстры, A-стар или Theta*, какой алгоритм запускается, указывается в входном .xml файле см. Приложение 1)

3) Создание выходного xml файла

3.3.2. Формат входных и выходных данных

Формат входного и выходного файла .xml приводится в приложении 1. Для запуска программы необходимо в аргументе командной строки передать путь до входного файла. Рассмотрим консольный вывод. В случае, если программа не может получить доступ к файлу, указанному в аргументе командной строки, будет выведено "Error! Pathfinding task file (XML) is not specified!", в случае успеха будет выведено "Parsing the map from XML:". В случае, если из входного файла невозможно выделить карту будет выведено "Incorrect map! Program halted!", в случае если невозможно определить конфигурацию запуска алгоритма будет выведено "Incorrect configurations! Program halted!", в случае успеха будет выведено соответственно "Map OK!", "Parsing configurations (algorithm, log) from XML:" и "Configurations OK!", "Creating log channel:". В случае если не удасться создать выходной файл будет выведено "Log chanel has not been created! Program halted!", в обратном случае "Log OK!", "Start searching the path:". Начнётся поиск пути, после его окончания будет выведено "Search is finished!". Если путь найден, то будет выведено "Path found", и, если путь не найден, "Path not found!". Также будет выведена длинна пути (если найден), количество шагов, сделанных алгоритмом, кол-во созданных вершин, время работы алгоритма.

3.3.2. Считывание данных из xml файла

Считывание данных из xml файла производится в соответствие с их форматом (см приложение 1). Адрес и имя файла передаётся в аргументах командной строки.

3.2.2.1. Алгоритм Дейкстры

Основная функция алгоритма.

Поиск пути (последовательности связных вершин, где первая вершина - стартовая, а последняя - финальная) в графе.

Алгоритм.

Будем называть стартовую вершину s, а конечную вершину f, а вычисленное расстояние от вершины s до вершины v будем обозначать как F[v].

Будем считать dist(v_1, v_2) как евклидово расстояние между центрами клеток v_1 и v_2.

Заведем список opened, который будет содержать все “открытые” вершины k и информацию о них (расстояние от стартовой вершины до этой вершины(F[k]), и вершина, из которой она была добавлена, будем называет её родительской по отношению к этой клетке, будем называть её p[k]), изначально положим в этот список стартовую вершину, расстояние до неё будет нулевым.

Заведем список closed, которой будет содержать все уже рассмотренные вершины и информацию о них (аналогичную информации в списке opened).

На каждом шаге извлекаем из списка opened вершину v с минимальным расстоянием от начальной вершины среди всех вершин списка opened, для каждой проходимой соседней(клетка, каждая координата которой отличается не более чем на 1 от рассматриваемой клетки) клетки n, не лежащей в closed, но принадлежащей opened, обновим значение d[n] = минимум(F[v] + расстояние от v до n, F[n]) , если в ходе этой операции F[n] изменилось, то обновим p[n] - теперь это вершина v, все проходимые соседние клетки k, не принадлежащие спискам closed и opened добавим в список opened, причем p[k] = v, p[] Далее заносим вершину v в список closed, переходим к следующему шагу.

Критерий остановки.

Завершаем вышеописанный алгоритм тогда, когда на i-ом шаге извлекаем из списка opened f или если прошло список opened стал пустым. Переходим к восстановлению пути в первом случае, во втором случае – пути не существует.

Алгоритм восстановления пути.

Извлечём из списка closed родительскую для f клетку p, добавим её в начало списка “путь”.

Повторим такую же процедуру для клетки p и т.д, пока не наткнёмся на клетку s.

Доказательство.

Докажем по индукции утверждение, что если из списка opened извлечена вершина, то расстояние до неё посчитано корректно (значение пути минимально). (*)

База индукции.

На момент извлечение стартовой вершины из списка opened, расстояние до неё посчитано корректно.

Переход индукции.

Пусть первые n шагов были проделаны корректно. На n + 1 ый шаг из списка opened извлекается вершина v. Докажем, что расстояние до неё посчитано корректно.

Предположим обратное. Рассмотрим путь из вершины s до вершины v. Возьмем вершину из этого пути, предшествующую вершине v. Если расстояние до p[v] посчитано корректно, то расстояние до v тоже должно быть посчитано корректно (так как когда мы извлекали вершину p[v], мы обновили расстояние до v, а значение расстояние до v не могло получиться меньше кратчайшего, так как каждому обновлению расстоянию сопоставлен путь по вершинам), значит, что расстояние до p посчитано некорректно, аналогично рассмотрим вершины из оптимального пути, предшествующую p, и далее, получим, что расстояние до всех вершин из кратчайшего пути было посчитано некорректно, но расстояние до изначальной вершины было посчитано корректно (!?). Следовательно, предположение (*) верно.

Пусть n – кол-во клеток. Докажем, что кол-во шагов алгоритма не более n.

Заметим, что на каждом шаге хотя бы одна вершина добавляется в список closed. Если все вершины были добавлены в список closed, то все вершины были извлечены из списка opened, что означает, что конечная вершина была извлечена из списка opened, и выполниться критерий остановки, следовательно кол-во шагов не более n.

Оценим асимптотику, с которой работает каждый шаг. Каждый шаг содержит константное кол-во обращений к спискам opened и closed, тогда каждый шаг имеет асимптотику log n. Списки opened и closed реализуются как декартовы деревья (описание ниже).

Асимптотика работы алгоритма – O(n log n).

3.2.2.2. Алгоритм A-стар

Основная функция алгоритма.

Алгоритм A-стар является расширенной версией алгоритма Дейкстры. Отличие от алгоритма Дейкстры в том, что в мы будем сперва рассматривать такие вершины, которые, как мы можем предположить, наиболее вероятно являются составляющими кратчайшего пути. Таким образом, в большом кол-ве случаев мы можем заметно сократить время работы алгоритма.

Алгоритм.

Введем понятие эвристической функции H[v] от вершины v, H[v] вычисляется между вершинами v и вершиной f по следующим правилам.

Эвристики (пусть x – разница двух вершин по x координате, y – по y координате):

1) Евклидова: √ (x * x + y * y).

2) Чебышева: максимум(x, y).

3) Манхэттенская: (x + y).

4) Диагональная: x + y – (2 - √2) * минимум(x, y).

Необходимое условие, для эвристики, для нахождения кратчайшего пути алгоритмом:

1) Произведение эвристического расстояния на вес эвристики не должно превышать возможное расстояние до данной вершины.

2) Должно соблюдаться правило треугольника (эвристический путь от a до b не должен быть больше, чем сумма эристических путей от a до c и от c до b).

Алгоритм не гарантирует нахождение кратчайшего пути, если эвристическое расстояние не удовлетворяет необходимому условию, но при этом он гарантирует нахождение пути, при условии его существования.

Алгоритм A-стар отличается от алгоритма Дейкстры тем, что вершины из списка opened извлекаются с минимальным значением F[v] + H[v] * вес эвристики.

Доказательство.

Докажем корректность работы алгоритма для корректной эвристики. Для начало докажем следующую лемму.

Лемма.

Для каждой вершины n и для каждого кратчайшего пути P из s в n существует в списке opened вершина p из P, такая, что F[p] равно кратчайшему расстоянию от s до p. F[p] – посчитанная в данный момент длинна пути.

Доказательство.

Пусть $P = \{n_0, n_1, \dots, n_k\}$. Назовем вершины из пути P, находящиеся в списке closed, как U. Возьмём вершину с максимальным индексом из множества U (оно не пусто, так как там лежит вершина s), пусть это вершина n_i , рассмотрим соседа n_i, вершину n_k из множества P(k > i). $F[n_k] <= F[n_i] + dist(n_i, n_k)$ (потому что когда мы рассматривали n_i мы пытались обновить значение $F[n_k]$), но каждому F[m] ставиться в соответствие путь из s в m, длинны F[m], таким образом F[m] всегда >= длинна кратчайшего пути из s в m. Но так как вершины n_k и n_i лежат в P, причем k > i, то длинная кратчайшего пути из s в n_k равна $F[n_i] + dist(n_i, n_k)$. Получаем: $F[n_i] + dist(n_i, n_k) <= F[n_k] <= F[n_i] + dist(n_i, n_k)$, что значит, что $F[n_k]$ = длине кратчайшего

пути из s в n_k. Лемма доказана.

Теперь перейдем к доказательству корректности алгоритма. Для начало докажем, что если путь существует, то алгоритм найдёт какой-то путь. Рассмотрим какой-то путь P = {n_0, n_1, …. , n_k}. Так как, вершины связны между собой, то все они будут добавлены в список opened, следовательно путь найдётся.

Доказательство алгоритма.

Перейдем к доказательству корректности алгоритма. Допустим алгоритм работает некорректно и завершившись на вершине f, F[f] > длины кратчайшего пути. Тогда, по выше рассмотренной лемме, перед тем как мы достали вершину f из списка opened, в списке opened была вершина n_i из кратчайшего пути P = {s = n_0, n_1, …. , n_k = f}, такая что F[n_i] равно длине кратчайшего пути из s в n_i. Но так как F[f] больше длины кратчайшего пути из вершины s в f, F[n_i] < F[f], тогда мы должны были извлечь из списка opened вершину n_i а не вершину f. (?!)

Получаем, что алгоритм работает корректно.

3.2.2.3. Алгоритм Theta*

Основная функция алгоритма.

Можно заметить, что пути, которые находит A-стар выглядят нереалистично и нередко переоценивают расстояние между двумя клетками на плоскости, что происходит из-за того, что в алгоритме A-стар мы передвигаемся только между центрами соседних клеток. Алгоритм Theta* является модификацией алгоритма A-стар, устраняющей этот недостаток. Хоть алгоритм и не гарантирует нахождение кратчайшего пути на плоскости, но результаты его работы лучше приближают путь на плоскости чем алгоритм A-стар.

Алгоритм.

Алгоритм Theta* отличается от алгоритма A-стар тем, что во время рассматривания соседей n клетки v, мы проверим, возможно ли провести прямую, соединяющие центры клеток n и v, так, чтобы она не пересекала препятствие. Если это возможно, то мы попытаемся обновить(или записать, если оно еще не записано) значение F[n] суммой F[p[v]] + dist(p[v], n). В качестве эвристической функции можно брать только евклидово расстояние, иначе мы можем переоценить расстояние до вершины.

Приведем алгоритм, проверяющий, возможно ли провести прямую между вершинами t и k. Клетки t и k имеют следующие координаты i_t j_t и i_k j_k. Рассмотрим случай, когда i_t <= i_k и j_t <= j_k, остальные случаи можно свести к этому, например заменив j координату всех клеток на ширину поля минус j координата, таким образом мы получим зеркальное отображение

поля.

Вычислим уравнение прямой.

Мы считаем, что i и j это координаты правого верхнего угла клетки, тогда координаты центра это i-0.5 и j-0.5.

Угловой коэффициент: $K = (i_t - i_k) / (j_t - j_k)$

Если K = 2, то пройдемся по всем клеткам, лежащим по диагонали от t и проверим, проходимы ли они. Иначе:

Уравнение прямой: $Y = K * X + i_k - K * j_k + 0.5 * K - 0.5$

Изначально рассматриваемая клетка – t, с координатами (i_t, j_t).

На каждом шаге будем проверять, лежит ли правый верхний угол рассматриваемой клетке над прямой или под прямой(подставим значение j в уравнение прямой и сравним его со значением i, если значение i больше, то правый верхний угол над прямой, если меньше то под прямой), если правый верхний угол лежит над прямой, то перейдем в клетку с координатами(i, j + 1), если лежит под прямой то в клетку (i + 1, j).

Если на каком-то шаге рассматриваемая клетка была непроходима, то завершаем алгоритм.

Доказательство.

Если мы нашли какой-то путь, то он корректен, так как у нас ему сопоставлена последовательность вершин, по которым этот путь проходит.

Докажем, что если существует путь из s в t то алгоритм найдёт путь из s в t.
Рассмотрим путь из s в t. P(s, … , t). Так как вершины в этом пути связаны, то все они будут добавлены в список opened, а значит извлечены из него, следовательно путь найдётся.

3.2.2.4. Алгоритм списков OPENED и CLOSED

Основная функция алгоритма.

Для реализации класс OPENED мною было выбрано декартово дерево.

Декартово дерево – структура данных, являющаяся по ключу деревом поиска, а по приоритету бинарной кучей. Декартово дерево позволяет искать элементы по ключу, позволяет искать минимум, добавлять и извлекать элементы. Все операции с бинарным деревом можно описать через две функции: split и merge.

Алгоритм.

Для начало опишем структуру, которую хранит OPENED.

Treap имеет следующие поля:

- val – вся информация о клетке
- prior – приоритет
- left – указатель на левого сына
- right – указатель на правого сына
- fun – минимальное значение пары (F[v]; H[v]), в соответствии с параметром breakingties.

Node имеет следующие поля:

- i – i-координата
- j – j-координата
- F – F-значение клетки
- H – значение эвристической функции от данной клетки
- previous_i – i-координата родительской клетки
- previous_j – j-координата родительской клетки

Будем считать ключом пару координат (i, j). Будем считать, что для всех вершин выполняется условие, значение ключа в правом сыне больше, чем в левом.

При создании вершины будем полагать её приоритет равный случайному числу (std::rand()).

Приступим к описанию операций. Операция merge сливает два декартова (d1 и d2) дерева в одно и возвращает указатель на него. Необходимое условие: все ключи первого меньше ключей второго (key(d1) < key(d2)). Алгоритм: вершиной нового дерева будет вершина с наибольшим приоритетом, если это d1, то ее новым правым сыном теперь будет merge от её старого правого сына и дерева d2, то есть d1.r = merge(d1.r, d2), левый сын останется без изменений. Если это d2, то её новым левым сыном будет merge от её старого левого сына и дерева d1, то есть d2.l = merge(d2.l, d1). После этой операции необходимо обновить значение fun в вершине, то есть записать в него минимум из значений в сыновьях и значении в val.

Операция split делит декартово дерево, на два d_1 и d_2, так что ключи в одном строго меньше значения key, а в другом больше или равно (key(d_1) < key(d_2)). Алгоритм: посмотрим на вершину дерева h, если она должна принадлежать d_1, то запустим split от правого сына вершины h (она разделит дерево на g_1 и g_2). Тогда итоговые деревья это merge(h (без правого сына);g_1), и g_2. После этой операции необходимо обновить значение fun в вершине, то есть записать в него минимум из значений в сыновьях и значении в val.

Операция добавления элемента k. Создадим элемент v класса treap, где val равен k, значение prior выбрано случайно, указатели на сыновей равны NULL. Алгоритм: сделаем split(key(v)), теперь у нас три декартова дерева d1, v, d2. Сделаем merge(merge(d1, v), d2).

Операция удаления элемента v. Алгоритм h.split(key(v)), d2.split(key(v) + 1), merge(d1, d2.d2).

Заметим, что две вышеописанные операции можно реализовать более эффективно. Так для того, чтобы вставить элемент, рекурсивно спустимся по дереву поиску в то место, где должна стоять эта вершина v, параллельно обновляя значение fun. Спускаться будем следующим способом: если значение ключа в вершине v больше чем в h, то перейдем в её правого сына, если меньше, то в левого и рекурсивно запустимся от них. Остановимся тогда, когда приоритет v станет больше приоритета h. Сделаем split(h; key(v)), левое возвращенной поддерево – левый сын v, правое – правый. Подвесим v на место h.

Извлечение элемента будем делать следующий способом, рекурсивно спускаемся в бинарном дереве в поиске элемента v с соответствующим ключом, параллельно обновляем значение fun. Подвесим merge(v.l, v.r) на место v.

Рекурсивно спускаемся в бинарном дереве, пока не найдём значение с соответствующим ключом, параллельно обновляя fun. Как только мы нашли искомую вершину, обновляем значение в ней и обновляем fun.

Заметим, что на каждом шаге split и merge мы спускаемся в дереве на один уровень, тогда итоговая сложность этих функций есть высота дерева. Здесь доказано, почему высота не превышает log(n). http://neerc.ifmo.ru/wiki/index.php?title=%D0%94%D0%B5%D0%BA%D0%B0%D1%80%D1%82%D0%BE%D0%B2%D0%BE_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE

3.2.3. Запись данных в xml файл

Запись данных из xml файла производится в соответствие с их форматом (см приложение 1).

4. ТЕХНИКО-ЭКОНОМИЧЕСКИЕ ПОКАЗАТЕЛИ

4.1 ЭКОНОМИЧЕСКИЕ ПОКАЗАТЕЛИ

Расчет экономических показателей не предусмотрен.

4.2 Предполагаемая потребность

Предполагается использование программы в соответствие с функциональным назначением.

5. ИСТОЧНИКИ, ИСПОЛЬЗОВАННЫЕ ПРИ РАЗРАБОТКЕ

1. “Алгоритмы: построение и анализ”, Томас Кормен, Чарльз Эрик Лейзерсон, Рональд Линн Ривест, Клиффорд Штайн, MIT Press, 1989 г., 1312.
2. Описание алгоритма A-star, интернет источник, дата обращения 12.11.2018, режим доступа: свободный. URL: https://ru.wikipedia.org/wiki/A*
3. Описание эвристик для A-star, интернет источник, дата обращения 12.12.2018, режим доступа: свободный. URL: https://www.cs.cmu.edu/~maxim/files/tutorials/robschooltutorial_oct10.pdf
4. Описание алгоритма Theta*, интернет источник, дата обращение 05.02.2019. URL: https://arxiv.org/pdf/1401.3843.pdf
5. Описание алгоритмов для декартового дерева, интернет источник, дата обращения 12.12.2018, режим доступа: свободный. URL: http://neerc.ifmo.ru/wiki/index.php?title=%D0%94%D0%B5%D0%BA%D0%B0%D1%80%D1%82%D0%BE%D0%B2%D0%BE_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE

Приложение 1.

Здесь описан формат входного и выходного файлов. Входной файл должен быть формата .xml.

Приведём тэги и опишем, как составить входной файл.

- <map> обозначает начало блока, в котором описано начальное пространство.
- <width> находиться внутри блока, обозначенного <map>, отвечает за ширину (кол-во столбцов) поля
- <height> находиться внутри блока, обозначенного <map>, отвечает за высоту (кол-во строк) поля
- <startx> находиться внутри блока, обозначенного <map>, обозначает блок, задающий x – координату начальной вершины (положительное целое число).
- <starty> находиться внутри блока, обозначенного <map>, обозначает блок, задающий y – координату начальной вершины (положительное целое число).
- <finishx> находиться внутри блока, обозначенного <map>, обозначает блок, задающий x – координату конечной вершины (положительное целое число).
- <finishy> находиться внутри блока, обозначенного <map>, обозначает блок, задающий y – координату конечной вершины (положительное целое число).
- <grid> находиться внутри блока, обозначенного <map>, обозначает начало блока, задающего описания ячеек пространство.
- <row> находиться внутри блока, обозначенного <grid>, обозначает начало блока, в котором описан ряд поля, т.е. задана последовательность из 1 и 0, разделенных пробелами. Символ 1 в i – ом ряду j – ом столбце обозначает, что эта ячейка поля не проходима, символ 0 – проходима.
- <algorithm> обозначает начало блока, задающего параметры, с которыми запускается алгоритм.
- <searchtype> находится в блоке, заданном <algorithm>, определяет используемый алгоритм (“astar”, “dijkstra”, “theta”).
- <metrictype> находится в блоке, заданном <algorithm>, определяет используемую эвристику (“diagonal”, “manhattan”, “euclidean”, “chebyshev”).
- <breakingties> находится в блоке, заданном <algorithm>, определяет выбор вершины, при равенстве F + G (“g-max”, “f-max”).
- <hweight> находится в блоке, заданном <algorithm>, определяет вес эвристики (целое число).
- <allowdiagonal> находится в блоке, заданном <algorithm>, определяет разрешено ли ходить по диагоналям (“true”, “false”).
- <cutcorners> находится в блоке, заданном <algorithm>, определяет разрешено ли срезать углы (“true”, “false”).
- <allowsqueeze> находится в блоке, заданном <algorithm>, определяет разрешено ли просачиваться (“true”, “false”).
- <logpath> обозначает блок, в котором описан путь для выходного файла.

Выходной файл содержит информацию о пространстве и алгоритме, описанную в входном файле.

- <log> о том, как отработал алгоритм.

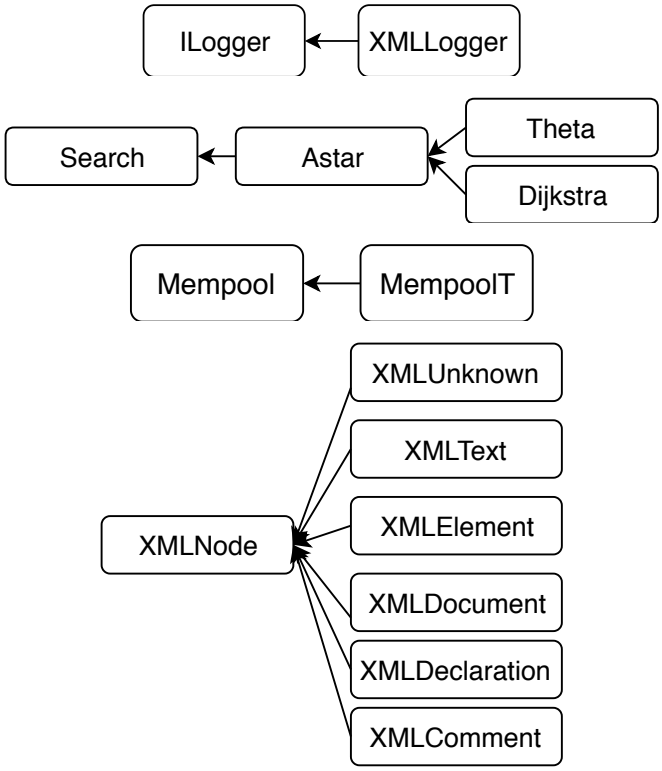
Перечисленные далее тэги лежат в <log>.

- <path> визуализировано поле (аналогично <grid>). Клетки, по которым пролегает оптимальный путь помечены как "*".
- <lplevel> по прядку перечисленные вершины, по которым пролегает оптимальный путь (<node x = “x - координата”, y = “y - координата”, number = “номер вершины”,>).
- <hplevel> по прядку перечисленные вершины, в которых меняется направление (аналогично <lplevel>).
- <summary> записано количество шагов, созданных вершин и длина пути

Каждый блок закрывается <\” тэг открывающий этот блок”>. (например, <map> и <\map>).

ПРИЛОЖЕНИЕ 2

Здесь описано наследование классов.



ПРИЛОЖЕНИЕ 3

Описание и функциональное назначение классов.

Класс	Назначение
Astar	Запуск алгоритма Astar
Config	Обработка параметров запуска алгоритма
Closed	Хранение вершинами
cmp_str	Упорядочивание вершин в соответствии с параметрами алгоритма
Dijkstra	Запуск алгоритма Dijkstra
EnvironmentOptions	Хранение параметров запуска алгоритма
ILogger	Запись выходные данных
ISearch	Поиск пути
Map	Хранение и работа с картой
Mission	Запуск алгоритма, работа с классами
Node	Хранение дополнительной информации о вершинах
SearchResult	Хранение результата поиска
Opened	Хранение вершин
treap	Хранение вершин декартового дерева
Theta	Запуск алгоритма Theta*
XmlLogger	Вывод результата работы алгоритма в выходной файл

ПРИЛОЖЕНИЕ 4

Описание и функциональное назначение полей методов и свойств.

Astar

1.1 Конструкторы:

Модиф. доступа	Название	Описание
public	Astar	Конструктор, передающий параметры поиска пути

Config

1.1 Конструкторы:

Модиф. доступа	Название	Описание
public	Config	Конструктор

1.2 Методы:

Модиф. доступа	Название	Описание
public	getConfig	Метод, конфигурирующий параметры запуска алгоритма и возвращающий 0 если возникли ошибки и 1 в обратном случае

1.3 Поля:

Модиф. доступа	Название	Описание
public	SearchParams	Параметры поиска
public	LogParams	Параметр выходного файла

public	N	Номер запущенного алгоритма

Closed

1.1 Конструкторы:

Модиф. доступа	**Название**	**Описание**
public	Closed	Конструктор, с параметром – ширина карты

1.2 Методы:

Модиф. доступа	**Название**	**Описание**
public	insert	Вставка элемента
public	find1	Поиск элемента
public	find	Проверка наличия элемента
public	output	Преобразование элементов для вывода

1.3 Поля:

Мо-диф. доступа	**Название**	**Описание**
private	vctr	Хранение вершин

cmp_str

1.1 Конструкторы:

Модиф. доступа	**Название**	**Описание**

public	cmp_str	Конструктор, с параметрами

1.2 Поля:

Модиф. доступа	**Название**	**Описание**
public	F	Функция раасстояния
public	H	Эвристическая функция
public	br	Параметр сравнения

Dijkstra

1.1 Конструкторы:

Модиф. доступа	**Название**	**Описание**
public	Dijkstra	Конструктор

EnvironmentOptions

1.1 Конструкторы:

Модиф. доступа	**Название**	**Описание**
public	EnvironmentOptions	Конструктор, с параметрами.

1.2 Поля:

Модиф. доступа	**Название**	**Описание**

public	metrictype	Эвристика
public	allowsqueeze	Разрешено ли просачивание
public	allowdiagonal	Разрешено ли ходить по диагонали
public	cutcorners	Разрешено ли срезать углы

ILogger

1.1 Конструкторы:

Модиф. доступа	**Название**	**Описание**
public	ILogger	Конструктор, с параметром

1.3 Методы:

Модиф. доступа	**Название**	**Описание**
public	getLog	Вставка элемента
public	saveLog	Поиск элемента
public	writeToLogMap	Вывод карты
public	writeToLogPath	Вывод пути
public	writeToLogOpenClose	Вывод списков opened и closed
public	writeToLogHPPath	Вывод пути по направлениям
public	writeToLogNotFound	Вывод отсутствия пути
public	writeToLogSummary	Вывод информации о работе алгоритма

1.4 Поля:

Модиф. доступа	**Название**	**Описание**
public	loglevel	Уровень лога

ISearch

1.1 Конструкторы:

Модиф. доступа	**Название**	**Описание**
public	ISearch	Конструктор, с параметрами

1.2 Методы:

Модиф. доступа	**Название**	**Описание**
protected	findSuccesors	Поиск соседей клетки
protected	calc_dist	Вычисление расстояние для соседей
protected	calc_distfortheta	Вычисление расстояние
protected	computeHFromCelltoCell	Вычисление эвристической функции
public	StartSearch	Запуск поиска
protected	make_hppath	Создание пути по направлениям
protected	is_parent	Проверка, может ли одна вершины быть родительской для другой
protected	ret_lp_parent	Определение всех клеток, по которым проходит путь
protected	make_hppath_theta	Создание пути по направлениям для Theta

protected	HF_Theta	Эвристика для Theta
protected	cmp	Сравнивает два Node
protected	new_fun	Обновление fun
protected	split	Разделение Декартового дерева на два
protected	merge	Сливание двух Декартовых деревьев
protected	erase	Извлечение элемента
protected	cmp1	Сравнение двух cmp_str
protected	find_min	Поиск минимума
protected	find	Проверка на существовании вершины с таким ключём
protected	find1	Поиск вершины с заданных ключём
protected	new_value	Попытка обновить вершины
protected	outp_value	Преобразование данных для вывода

1.3 Поля:

Модиф. доступа	**Название**	**Описание**
private	search_par	Параметр поиска
private	sresult	Результат поиска пути
private	lppath	Все вершины, через которые пролегает путь
private	hppath	Вершины, в которых меняется направление

private	hweight	Вес эвристики
private	breakingties	Параметр сортировки вершин

Map

1.1 Конструкторы:

Модиф. доступа	**Название**	**Описание**
public	Map	Конструктор

1.2 Методы:

Модиф. доступа	**Название**	**Описание**
public	getMap	Проверка корректности карты
public	CellIsTranversable	Проверка клетки на проходимость
public	CellIsOnGrid	Проверка клетки на принадлежность карте
public	CellIsObstacle	Проверка клетки на не проходимость

1.3 Поля:

Модиф. доступа	**Название**	**Описание**
private	height	Высота поля
private	width	Ширина поля
private	start_i	I-координата начальной вершины
private	start_j	J-координата начальной вершины

private	goal_i	I-координата конечной вершины
private	goal_j	J-координата конечной вершины
private	cellSize	Размер клетки
private	Grid	Карта

1.4 Свойства:

Модиф. доступа	**Название**	**Описание**
public	getValue	Преобразование элементов для вывода
public	getMapHeight	Преобразование элементов для вывода
public	getMapWidth	Преобразование элементов для вывода
public	getCellSize	Преобразование элементов для вывода
public	GetStart	Вставка элемента
public	GetEnd	Поиск элемента

Mission

1.1 Конструкторы:

Модиф. доступа	**Назва-ние**	**Описание**
public	Mission	Конструктор, с параметром – имя файла

1.4 Методы:

Модиф. доступа	**Название**	**Описание**
public	createLog	Создание выходного лога
public	createSearch	Создать поиск

public	createEnvironmentOptions	Создание конфигураций алгоритма
public	startSearch	Начать поиск пути
public	printSearchResultsToConsole	Вывести результат в консоль
public	saveSearchResultsToLog	Сохранить результаты в выходной файл
private	getAlgorithmName	Вернуть имя алгоритма

1.5 Поля:

Модиф. доступа	**Название**	**Описание**
private	map	Карта
private	config	Конфигурации алгоритма
private	options	Опции алгоритма
private	search	Поиск пути
private	logger	Лог
private	fileName	Имя файла
private	sr	Результаты поиска пути

1.4 Свойства:

Модиф. доступа	**Название**	**Описание**
public	getMap	Возврат карты
public	getConfig	Возврат конфигураций алгоритма

Node

1.1 Конструкторы:

Модиф. доступа	**Название**	**Описание**
public	Node	Конструктор, с параметрами

1.2 Поля:

Модиф. доступа	**Название**	**Описание**
public	i	I-координата вершины
public	j	J-координата вершины
public	previous_i	I-координата родительской вершины
public	previous_j	J-координата родительской вершины
public	F	Функция расстояния от начальной вершины
public	H	Эвристическая функция
public	brk	Параметр сравнение

Opened

1.1 Конструкторы:

Модиф. доступа	**Название**	**Описание**
public	Opened	Конструктор

1.2 Методы:

Модиф. доступа	**Название**	**Описание**
public	insert	Вставка элемента

public	erase	Извлечение элемента
public	erase_minimum	Извлечение минимума
public	minm	Вывод списка в консоль
public	new_value	Попытаться обновить значение
public	find	Проверка наличия элемента
public	find1	Вернуть найденный элемент
public	output	Преобразовать данные для вывода в консоль

1.3 Поля:

Мо-диф. доступа	**Название**	**Описание**
private	head	Указатель на корень дерева

1.4 Свойства:

Модиф. доступа	**Название**	**Описание**
public	notempty	Возвращает 1 если список не пуст

SearchResult

1.1 Конструкторы:

Модиф. доступа	**Название**	**Описание**
public	SearchResult	Конструктор

1.2 Поля:

Модиф. доступа	**Название**	**Описание**

public	pathfound	Переменная, показывающая, найден ла путь
public	pathlength	Длина пути
public	lppath	Все вершины, по которым проходит путь
public	hppath	Все вершины пути, в которых он меняет направление
public	nodescreated	Кол-во созданных вершин
public	numberofsteps	Кол-во шагов, сделанных алгоритмом
public	time	время работы алгоритма

treap

1.1 Поля:

Модиф. доступа	**Название**	**Описание**
public	val	Вершина и информация о ней
public	prior	Приоритет
public	l	Указатель на левого сына
public	r	Указатель на правого сына
public	fun	Минимальное значение по всем поддеревьям

Theta

1.3 Конструкторы:

Модиф. доступа	**Название**	**Описание**
public	Theta	Конструктор

ILogger

1.1 Конструкторы:

Модиф. доступа	**Название**	**Описание**
public	ILogger	Конструктор, с параметром

1.2 Методы:

Модиф. доступа	**Название**	**Описание**
public	getLog	Вставка элемента
public	saveLog	Поиск элемента
public	writeToLogMap	Вывод карты
public	writeToLogPath	Вывод пути
public	writeToLogOpenClose	Вывод списков opened и closed
public	writeToLogHPPath	Вывод пути по направлениям
public	writeToLogNotFound	Вывод отсутствия пути
public	writeToLogSummary	Вывод информации о работе алгоритма

1.3 Поля:

Модиф. доступа	**Название**	**Описание**
private	loglevel	Уровень лога
private	doc	Объект для выводы данных

ЛИСТ РЕГИСТРАЦИИ ИЗМЕНЕНИЙ

зм.	Номера листов (страниц)				Всего листов (страниц) в документе	№ документа	Входящий № сопроводительного документа и дата	Подпись	Дата
	измененных	замененных	новых	аннулированных					

Изм.	Лист	№ докум.	Подп.	Дата
RU.17701729.04.01-01 81				
Инв. № подл.	Подп. и дата	Взам. инв. №	Инв. № дубл.	Подп. и дата