**МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ**

**Ордена Трудового Красного Знамени**

**Федеральное государственное бюджетное образовательное учреждение высшего образования**

**«Московский технический университет связи и информатики»**

Кафедра «Математическая кибернетика и информационные технологии»

**Отчет по лабораторной работе №3**

по дисциплине «Информационные технологии и программирование»

на тему:

Алгоритм A\* («A star»)

Выполнил: студент группы БФИ2101

Иванов Глеб

Проверил преподаватель:

Мосева Марина Сергеевна

Москва, 2022

Оглавление

[**Поставленная задача** 3](#_Toc122867251)

[**Скриншоты выполненного задания** 6](#_Toc122867252)

[**Код** 7](#_Toc122867253)

[**Вывод** 30](#_Toc122867254)

[**Список источников** 31](#_Toc122867255)

**Поставленная задача**

**Локации**

Для начала необходимо подготовить класс Location для совместного использования с классами коллекции Java. Поскольку вы будете использовать контейнеры для хеширования для выполнения данного задания, то для этого необходимо:

• Обеспечить реализацию метода equals ().

• Обеспечить реализацию метода hashcode().

Добавьте реализацию каждого из этих методов в класс Location, следуя шаблонам в классе. После этого вы можете использовать класс Location в качестве ключевого типа в контейнерах хеширования, таких как HashSet и HashMap.

Состояния А\* После того, как класс Location готов к использованию, вы можете завершить реализацию класса AStarState. Это класс, который поддерживает наборы открытых и закрытых вершин, поэтому он действительно обеспечивает основную функциональность для реализации алгоритма А\*.

Как упоминалось ранее, состояние А\* состоит из двух наборов вершин, один из открытых вершин и другой из закрытых. Чтобы упростить алгоритм, вершины будут храниться в хэш-карте, где местоположение вершин является ключом, а сами вершины являются значениями. Таким образом, у вас будет такой тип: HashMap (Очевидный вывод из всего этого заключается в том, что с каждым местоположением на карте может быть связана только одна вершина.)

Добавьте два (нестатических) поля в класс AStarState с таким типом, одно для "открытых вершин" и другой для "закрытых вершин". Кроме того, не забудьте инициализировать каждое из этих полей для ссылки на новую пустую коллекцию. После создания и инициализации полей, вы должны реализовать следующие методы в классе AStarState:

1) public int numOpenWaypoints() Этот метод возвращает количество точек в наборе открытых вершин.

2) public Waypoint getMinOpenWaypoint() Эта функция должна проверить все вершины в наборе открытых вершин, и после этого она должна вернуть ссылку на вершину с наименьшей общей стоимостью. Если в "открытом" наборе нет вершин, функция возвращает NULL. Не удаляйте вершину из набора после того, как вы вернули ее; просто верните ссылку на точку с наименьшей общей стоимостью.

3) public boolean addOpenWaypoint(Waypoint newWP) Это самый сложный метод в классе состояний А\*. Данный метод усложняет то, что он должен добавлять указанную вершину только в том случае, если существующая вершина хуже новой. Вот что должен делать этот метод:

• Если в наборе «открытых вершин» в настоящее время нет вершины для данного местоположения, то необходимо просто добавить новую вершину.

• Если в наборе «открытых вершин» уже есть вершина для этой локации, добавьте новую вершину только в том случае, если стоимость пути до новой вершины меньше стоимости пути до текущей. (Убедитесь, что используете не общую стоимость.) Другими словами, если путь через новую вершину короче, чем путь через текущую вершину, замените текущую вершину на новую Как вы могли заметить, что в таком случае вам потребуется извлечь существующую вершину из «открытого набора», если таковая имеется. Данный шаг довольно прост - замените предыдущую точку на новую, используя метод HashMap.put(), который заменит старое значение на новое. Пусть данный метод вернет значение true, если новая вершина была успешно добавлена в набор, и false в противном случае.

4) public boolean isLocationClosed(Location loc) Эта функция должна возвращать значение true, если указанное местоположение встречается в наборе закрытых вершин, и false в противном случае. Так как закрытые вершины хранятся в хэш-карте с расположениями в качестве ключевых значений, данный метод достаточно просто в реализации.

5) public void closeWaypoint(Location loc) Эта функция перемещает вершину из набора «открытых вершин» в набор «закрытых вершин». Так как вершины обозначены местоположением, метод принимает местоположение вершины. Процесс должен быть простым:

• Удалите вершину, соответствующую указанному местоположению из набора «открытых вершин».

• Добавьте вершину, которую вы удалили, в набор закрытых вершин. Ключом должно являться местоположение точки.

**Компиляция и тестирование**

Как только вы реализуете вышеуказанную функциональность, запустите программу поиска пути, чтобы проверит правильность ее выполнения. Если вы реализовали все правильно, то у вас не должно возникнуть проблем при создании препятствий и последующим поиском путей вокруг них. Скомпилируйте и запустите программу также, как и всегда: javac \*.java java AStarApp

**Скриншоты выполненного задания**

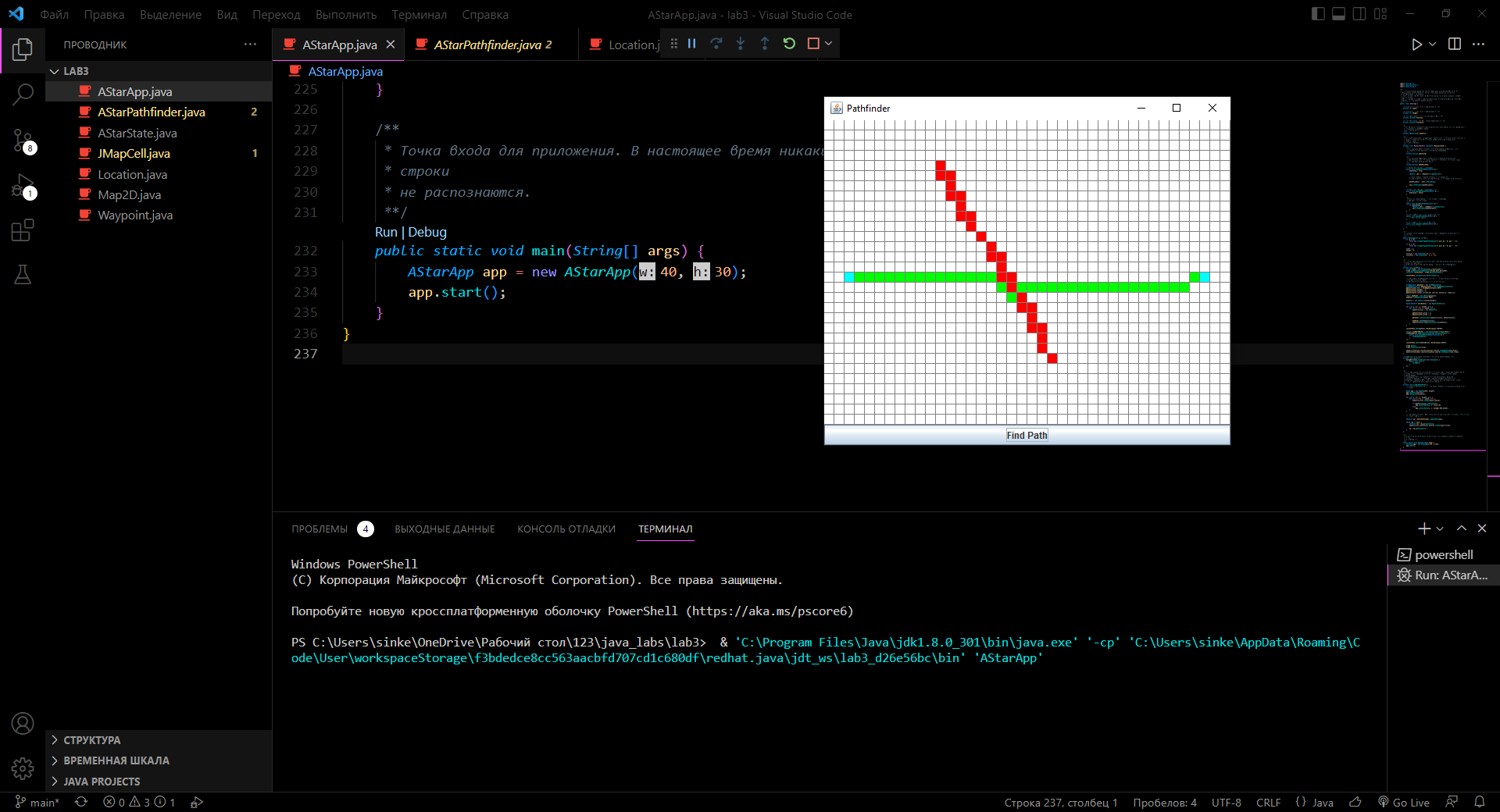


Рисунок 1 - Выполнение поставленной задачи

**Код**

import *java*.*awt*.\*;

import *java*.*awt*.*event*.\*;

import *javax*.*swing*.\*;

/\*\*

*\* Простое приложение Swing для демонстрации алгоритма поиска пути A \*. То*

*\* пользователю предоставляется карта, содержащая начальное и конечное*

*\* местоположение. Пользователь*

*\* может рисовать или устранять препятствия на карте, а затем нажимать кнопку*

*\* для вычисления*

*\* путь от начала до конца с использованием алгоритма поиска пути A\*. Если путь*

*\* найден, он отображается зеленым цветом.*

*\**\*/

*public* *class* AStarApp {

/\*\* *Количество ячеек сетки в направлении X. \**\*/

*private* *int* width;

/\*\* *Количество ячеек сетки в направлении Y. \**\*/

*private* *int* height;

/\*\* *Местоположение, с которого начинается путь. \**\*/

*private* *Location* startLoc;

/\*\* *Местоположение, где путь должен заканчиваться. \**\*/

*private* *Location* finishLoc;

/\*\*

*\* Это 2D-массив компонентов пользовательского интерфейса, которые обеспечивают*

*\* отображение и манипулирование*

*\* из ячеек на карте.*

*\*\**\*/

*private* *JMapCell*[][] mapCells;

/\*\*

*\* Этот внутренний класс обрабатывает события мыши в основной сетке ячеек карты,*

*\* изменяя ячейки на основе состояния кнопки мыши и первоначального*

*\* редактирования.*

*\* это было выполнено.*

*\**\*/

*private* *class* MapCellHandler *implements* MouseListener {

/\*\*

*\* Это значение будет истинным, если была нажата кнопка мыши, и мы*

*\* в данный момент находимся в процессе модификации.*

*\**\*/

*private* *boolean* modifying;

/\*\*

*\* Это значение записывает, делаем ли мы ячейки проходимыми или*

*\* непроходимый. Что это такое, зависит от исходного состояния ячейки*

*\* что операция была начата внутри.*

*\**\*/

*private* *boolean* makePassable;

        // *Инициирует операцию модификации.*

*public* *void* mousePressed(*MouseEvent* e) {

            modifying = true;

*JMapCell* cell = (JMapCell) e.getSource();

            // *Если текущая ячейка проходима, то мы делаем их*

            // *непроходимыми; если она непроходима, то мы делаем их проходимыми.*

            makePassable = !cell.isPassable();

            cell.setPassable(makePassable);

        }

/\*\* *Завершает операцию модификации. \**\*/

*public* *void* mouseReleased(*MouseEvent* e) {

            modifying = false;

        }

/\*\*

*\* Если мышь была нажата, это продолжает модификацию*

*\* операция в новой ячейке.*

*\**\*/

*public* *void* mouseEntered(*MouseEvent* e) {

            if (modifying) {

*JMapCell* cell = (JMapCell) e.getSource();

                cell.setPassable(makePassable);

            }

        }

/\*\* *Не требуется для этого обработчика. \**\*/

*public* *void* mouseExited(*MouseEvent* e) {

            // *This one we ignore.*

        }

        // *Не требуется для этого обработчика.*

*public* *void* mouseClicked(*MouseEvent* e) {

            // *И этот тоже.*

        }

    }

/\*\*

*\* \* Создает новый экземпляр приложения Star с указанной шириной карты и*

*\* высотой \*.*

*\**\*/

*public* AStarApp(*int* w, *int* h) {

        if (w <= 0)

            throw new *IllegalArgumentException*("w must be > 0; got "+ *w*);

        if (h <= 0)

            throw new *IllegalArgumentException*("h must be > 0; got "+ *h*);

        width = w;

        height = h;

        startLoc = new *Location*(2, *h* /2);

        finishLoc = new *Location*(*w* -3, *h* /2);

    }

/\*\*

*\* Простой вспомогательный метод для настройки пользовательского интерфейса*

*\* Swing. Это называется*

*\* из потока обработчика событий Swing, чтобы быть потокобезопасным.*

*\**\*/

*private* *void* initGUI() {

*JFrame* frame = new *JFrame*("Pathfinder");

        frame.setDefaultCloseOperation(JFrame.*EXIT\_ON\_CLOSE*);

*Container* contentPane = frame.getContentPane();

        contentPane.setLayout(new *BorderLayout*());

        // *Используйте GridBagLayout, потому что он действительно учитывает*

        // *предпочтительный размер*

        // *определяется компонентами, которые он содержит.*

*GridBagLayout* gbLayout = new *GridBagLayout*();

*GridBagConstraints* gbConstraints = new *GridBagConstraints*();

        gbConstraints.*fill* = GridBagConstraints.*BOTH*;

        gbConstraints.*weightx* = 1;

        gbConstraints.*weighty* = 1;

        gbConstraints.*insets*.set(0, 0, 1, 1);

*JPanel* mapPanel = new *JPanel*(*gbLayout*);

        mapPanel.setBackground(Color.*GRAY*);

        mapCells = new *JMapCell*[width][height];

*MapCellHandler* cellHandler = new *MapCellHandler*();

        for (*int* y = 0; y < height; y++) {

            for (*int* x = 0; x < width; x++) {

                mapCells[x][y] = new *JMapCell*();

                gbConstraints.*gridx* = x;

                gbConstraints.*gridy* = y;

                gbLayout.setConstraints(mapCells[x][y], gbConstraints);

                mapPanel.add(mapCells[x][y]);

                mapCells[x][y].addMouseListener(cellHandler);

            }

        }

        contentPane.add(mapPanel, BorderLayout.*CENTER*);

*JButton* findPathButton = new *JButton*("Find Path");

        findPathButton.addActionListener(new *ActionListener*() {

*public* *void* actionPerformed(*ActionEvent* e) {

*findAndShowPath*();

            }

        });

        contentPane.add(findPathButton, BorderLayout.*SOUTH*);

        frame.pack();

        frame.setVisible(true);

        mapCells[startLoc.*xCoord*][startLoc.*yCoord*].setEndpoint(true);

        mapCells[finishLoc.*xCoord*][finishLoc.*yCoord*].setEndpoint(true);

    }

/\*\* *Запускает приложение. Вызывается из метода {@link #main}. \**\*/

*private* *void* start() {

        SwingUtilities.invokeLater(new *Runnable*() {

*public* *void* run() {

*initGUI*();

            }

        });

    }

/\*\*

*\* Этот вспомогательный метод пытается вычислить путь, используя текущую карту*

*\* государство. Реализация довольно медленная; создается новый объект*

*\* {@link Map2D}*

*\* и инициализируется из текущего состояния приложения. Затем А\**

*\* вызывается навигатор, и если путь найден, дисплей обновляется до*

*\* показывать найденный путь. (Лучшим решением было бы использовать модель*

*\* Просмотр шаблона проектирования контроллера.)*

*\**\*/

*private* *void* findAndShowPath() {

        // *Создайте 2D-объект карты, содержащий текущее состояние пользовательского*

        // *ввода.*

*Map2D* map = new *Map2D*(*width*, *height*);

        map.setStart(startLoc);

        map.setFinish(finishLoc);

        for (*int* y = 0; y < height; y++) {

            for (*int* x = 0; x < width; x++) {

                mapCells[x][y].setPath(false);

                if (mapCells[x][y].isPassable())

                    map.setCellValue(x, y, 0);

                else

                    map.setCellValue(x, y, Integer.*MAX\_VALUE*);

            }

        }

        // *Попробуйте вычислить путь. Если один из них может быть вычислен, отметьте все*

        // *ячейки в пути //.*

*Waypoint* wp = AStarPathfinder.computePath(map);

        while (wp != null) {

*Location* loc = wp.getLocation();

            mapCells[loc.*xCoord*][loc.*yCoord*].setPath(true);

            wp = wp.getPrevious();

        }

    }

/\*\*

*\* Точка входа для приложения. В настоящее время никакие аргументы командной*

*\* строки*

*\* не распознаются.*

*\**\*/

*public* *static* *void* main(*String*[] args) {

*AStarApp* app = new *AStarApp*(40,30);

        app.start();

    }

}

**Код файла** AStarApp.java

import *java*.*util*.*HashMap*;

import *java*.*util*.*HashSet*;

/\*\*

*\* Этот класс содержит реализацию алгоритма поиска пути A\*. Алгоритм*

*\* реализован как статический метод, поскольку алгоритм поиска пути*

*\* на самом деле не нужно поддерживать какое-либо состояние между вызовами*

*\* алгоритма*

*\* .*

\*/

*public* *class* AStarPathfinder {

/\*\*

*\* Эта константа содержит максимальный предел отсечения для стоимости путей.*

*\* Если бы*

*\* конкретная путевая точка превышает этот лимит затрат, путевая*

*\* точка \* отбрасывается.*

*\**\*/

*public* *static* *final* *float* COST\_LIMIT = 1e6f;

/\*\*

*\* Пытается вычислить путь, который перемещается между начальным и конечным*

*\* местоположениями указанной карты. Если путь может быть найден, возвращается*

*\* путевая точка*

*\* последнего </em> шага пути; эта путевая точка может быть*

*\* использована для обратного перехода к начальной точке. Если путь не может*

*\* быть найден,*

*\* возвращается \* <код>null</code>.*

*\**\*/

*public* *static* *Waypoint* computePath(*Map2D* map) {

        // *Переменные, необходимые для поиска A\*.*

*AStarState* state = new *AStarState*(*map*);

*Location* finishLoc = map.getFinish();

        // *Установите начальную путевую точку, чтобы начать поиск A\*.*

*Waypoint* start = new *Waypoint*(map.*getStart*(), *null*);

        start.setCosts(0, *estimateTravelCost*(start.*getLocation*(), *finishLoc*));

        state.addOpenWaypoint(start);

*Waypoint* finalWaypoint = null;

*boolean* foundPath = false;

        while (!foundPath && state.numOpenWaypoints() > 0) {

            // *Найдите "лучшую" (т.е. самую дешевую) путевую точку на данный момент.*

*Waypoint* best = state.getMinOpenWaypoint();

            // *Если лучшее место - это место финиша, то мы закончили!*

            if (best.getLocation().equals(finishLoc)) {

                finalWaypoint = best;

                foundPath = true;

            }

            // *Добавить / обновить всех соседей текущего наилучшего местоположения. Это*

            // *эквивалентно попытке выполнить все "следующие шаги" из этого местоположения.*

*takeNextStep*(*best*, *state*);

            // *Наконец, переместите это местоположение из списка "открыто" в список*

            // *"закрыто"*

            // *список.*

            state.closeWaypoint(best.getLocation());

        }

        return finalWaypoint;

    }

/\*\*

*\* Этот статический вспомогательный метод принимает путевую точку и генерирует*

*\* все допустимые "далее*

*\* шаги" от этой точки маршрута. Новые путевые точки добавляются в "открытый*

*\* путевые точки" коллекция переданного объекта состояния\*.*

*\**\*/

*private* *static* *void* takeNextStep(*Waypoint* currWP, *AStarState* state) {

*Location* loc = currWP.getLocation();

*Map2D* map = state.getMap();

        for (*int* y = loc.*yCoord* - 1; y <= loc.*yCoord* + 1; y++) {

            for (*int* x = loc.*xCoord* - 1; x <= loc.*xCoord* + 1; x++) {

*Location* nextLoc = new *Location*(*x*, *y*);

                // *Если "следующее местоположение" находится за пределами карты, пропустите его.*

                if (!map.contains(nextLoc))

                    continue;

                // *Если "следующее местоположение" - это это местоположение, пропустите его.*

                if (nextLoc == loc)

                    continue;

                // *Если это местоположение уже находится в "закрытом" наборе*

                // *затем перейдите к следующему местоположению.*

                if (state.isLocationClosed(nextLoc))

                    continue;

                // *Сделайте путевую точку для этого "следующего местоположения".*

*Waypoint* nextWP = new *Waypoint*(*nextLoc*, *currWP*);

                // *Хорошо, мы жульничаем и используем смету затрат для вычисления фактического*

                // *стоимость из предыдущей ячейки. Затем мы добавляем стоимость из*

                // *// ячейка карты, в которую мы входим, чтобы включить барьеры и т.д.*

*float* prevCost = currWP.getPreviousCost() +

*estimateTravelCost*(currWP.*getLocation*(),

nextWP.*getLocation*());

                prevCost += map.getCellValue(nextLoc);

                // *Пропустите это "следующее местоположение", если это слишком дорого.*

                if (prevCost >= COST\_LIMIT)

                    continue;

                nextWP.setCosts(prevCost,

*estimateTravelCost*(*nextLoc*,map.*getFinish*()));

                // *Добавьте путевую точку в набор открытых путевых точек. Если там*

                // *уже является путевой точкой для этого местоположения, новая*

                // *путевая точка заменяет старую путевую точку только в том случае, если это*

                // *менее затратно*

                // *чем старый.*

                state.addOpenWaypoint(nextWP);

            }

        }

    }

/\*\*

*\* Оценивает стоимость проезда между двумя указанными точками.*

*\* Рассчитанная фактическая стоимость - это просто расстояние по прямой между*

*\* двумя местоположениями.*

*\**\*/

*private* *static* *float* estimateTravelCost(*Location* currLoc, *Location* destLoc) {

*int* dx = destLoc.*xCoord* - currLoc.*xCoord*;

*int* dy = destLoc.*yCoord* - currLoc.*yCoord*;

        return (*float*) Math.sqrt(dx \* dx + dy \* dy);

    }

}

**Код** AStarPathfinder.java

import *java*.*util*.*HashMap*;

/\*\*

*\* Этот класс хранит базовое состояние, необходимое для алгоритма A\* для*

*\* вычисления*

*\* путь по карте. Это состояние включает в себя коллекцию "открытых путевых*

*\* точек" и*

*\* другую коллекцию "закрытых путевых точек". Кроме того, этот класс*

*\* обеспечивает*

*\* основные операции, необходимые алгоритму поиска пути A\* для выполнения его*

*\* обработка.*

*\**\*/

*public* *class* AStarState {

/\*\* *Это ссылка на карту, по которой перемещается алгоритм A\*. \**\*/

*private* *Map2D* map;

*private* *final* *HashMap*<*Location*, *Waypoint*> closedWaypoints = new *HashMap*<>();

*private* *final* *HashMap*<*Location*, *Waypoint*> openedWaypoints = new *HashMap*<>();

/\*\*

*\* Инициализируйте новый объект состояния для использования алгоритма поиска*

*\* пути A\*.*

*\**\*/

*public* AStarState(*Map2D* map) {

        if (map == null)

            throw new *NullPointerException*("map cannot be null");

        this.*map* = map;

    }

/\*\* *Возвращает карту, по которой перемещается навигатор A\*. \**\*/

*public* *Map2D* getMap() {

        return map;

    }

/\*\*

*\* Этот метод сканирует все открытые путевые точки и возвращает путевую точку*

*\* с минимальной общей стоимостью. Если открытых путевых точек нет, этот метод*

*\* возвращает <код>null</code>.*

*\**\*/

*public* *Waypoint* getMinOpenWaypoint() {

*Location* minLocation = null;

*float* minTotalCost = Float.*MAX\_VALUE*;

        for (*Location* key : openedWaypoints.keySet()) {

*float* totalCost = openedWaypoints.get(key).getTotalCost();

            if (totalCost < minTotalCost) {

                minTotalCost = totalCost;

                minLocation = key;

            }

        }

        return openedWaypoints.get(minLocation);

    }

/\*\*

*\* Этот метод добавляет путевую точку к (или потенциально обновляет уже*

*\* имеющуюся путевую точку*

*\* в) коллекция "открытые путевые точки". Если еще не существует открытого*

*\* \* путевая точка в новом местоположении путевых точек, тогда новая путевая*

*\* точка просто*

*\* добавлено в коллекцию. Однако, если в*

*\* местоположении \* \* new waypoints уже есть путевая точка, новая путевая точка*

*\* заменяет только старую <em>*

*\* \* если</em> значение "предыдущей стоимости" новых путевых точек меньше*

*\* текущего*

*\* \* значение путевых точек "предыдущая стоимость".*

*\**\*/

*public* *boolean* addOpenWaypoint(*Waypoint* newWP) {

*Location* location = newWP.getLocation();

        if (!openedWaypoints.containsKey(location)

                || newWP.getPreviousCost() < openedWaypoints.get(location).getPreviousCost()) {

            openedWaypoints.put(location, newWP);

            return true;

        }

        return false;

    }

/\*\* *Возвращает текущее количество открытых путевых точек. \**\*/

*public* *int* numOpenWaypoints() {

        // *TODO: Implement.*

        return openedWaypoints.size();

    }

/\*\*

*\* Этот метод перемещает путевую точку в указанном местоположении из*

*\* открытого списка в закрытый список.*

*\**\*/

*public* *void* closeWaypoint(*Location* loc) {

*Waypoint* wpToClose = openedWaypoints.remove(loc);

        if (wpToClose != null) {

            closedWaypoints.put(loc, wpToClose);

        }

    }

/\*\*

*\* Возвращает значение true, если коллекция закрытых путевых точек содержит*

*\* путевую точку*

*\* для указанного местоположения.*

*\**\*/

*public* *boolean* isLocationClosed(*Location* loc) {

        return closedWaypoints.containsKey(loc);

    }

}

**Код файла** AStarState.java

import *java*.*awt*.\*;

import *javax*.*swing*.\*;

import *javax*.*swing*.*border*.\*;

/\*\*

*\* Этот класс является пользовательским компонентом Swing для представления*

*\* одной ячейки карты в*

*\* 2D-карта. Ячейка имеет несколько различных видов состояния, но самое основное*

*\* состояние - это то, является ли ячейка проходимой или нет.*

\*/

*public* *class* JMapCell *extends* JComponent {

*private* *static* *final* *Dimension* CELL\_SIZE = new *Dimension*(12,12);

/\*\*

*\* True указывает, что ячейка является конечной точкой, либо начальной, либо*

*\* конечной.*

*\**\*/

*boolean* endpoint = false;

/\*\*

*\* Значение True указывает на то, что ячейка проходима; значение false означает,*

*\* что это не так.*

*\**\*/

*boolean* passable = true;

/\*\*

*\* True указывает, что эта ячейка является частью пути между началом и концом.*

*\**\*/

*boolean* path = false;

/\*\*

*\* \* Создайте новую ячейку карты с заданной "проходимостью". Ввод значения*

*\* true означает, что ячейка проходима.*

*\**\*/

*public* JMapCell(*boolean* pass) {

        // *Установите предпочтительный размер ячейки, чтобы задать начальный размер*

        // *окна.*

*setPreferredSize*(*CELL\_SIZE*);

*setPassable*(*pass*);

    }

/\*\* *Создайте новую ячейку карты, которая по умолчанию является проходимой. \**\*/

*public* JMapCell() {

        // *Вызовите другой конструктор, указав значение true для "passable".*

        this(true);

    }

/\*\* *Помечает эту ячейку либо как начальную, либо как конечную. \**\*/

*public* *void* setEndpoint(*boolean* end) {

        endpoint = end;

*updateAppearance*();

    }

/\*\*

*\* Устанавливает эту ячейку проходимой или непроходимой. Ввод истинных меток*

*\* ячейка как проходимая; ввод значения false помечает ее как непроходимую.*

*\**\*/

*public* *void* setPassable(*boolean* pass) {

        passable = pass;

*updateAppearance*();

    }

/\*\*

*\* Возвращает true, если эта ячейка проходима, или false в противном случае.*

*\**\*/

*public* *boolean* isPassable() {

        return passable;

    }

/\*\* *Переключает текущее "проходимое" состояние ячейки карты. \**\*/

*public* *void* togglePassable() {

*setPassable*(!*isPassable*());

    }

/\*\* *Помечает эту ячейку как часть пути, обнаруженного алгоритмом A\*. \**\*/

*public* *void* setPath(*boolean* path) {

        this.*path* = path;

*updateAppearance*();

    }

/\*\*

*\* Этот вспомогательный метод обновляет цвет фона, чтобы он соответствовал*

*\* текущему*

*\* внутреннее состояние ячейки.*

*\**\*/

*private* *void* updateAppearance() {

        if (passable) {

            // *Проходимая ячейка. Укажите его состояние с помощью границы.*

*setBackground*(Color.*WHITE*);

            if (endpoint)

*setBackground*(Color.*CYAN*);

            else if (path)

*setBackground*(Color.*GREEN*);

        } else {

            // *Непроходимая клетка. Сделай все это красным.*

*setBackground*(Color.*RED*);

        }

    }

/\*\*

*\* Реализация метода paint для рисования цвета фона в ячейке*

*\* map.*

*\**\*/

*protected* *void* paintComponent(*Graphics* g) {

        g.setColor(*getBackground*());

        g.fillRect(0, 0, *getWidth*(), *getHeight*());

    }

}

**Код файла** JMapCell.java

import *java*.*util*.*Objects*;

/\*\*

*\* Этот класс представляет определенное местоположение на 2D-карте. Координаты*

*\* являются*

*\* целочисленные значения.*

*\**\*/

*public* *class* Location {

/\*\* *X координата этого местоположения. \**\*/

*public* *int* xCoord;

/\*\* *Y координата этого местоположения. \**\*/

*public* *int* yCoord;

/\*\* *Создает новое местоположение с указанными целочисленными координатами. \**\*/

*public* Location(*int* x, *int* y) {

        xCoord = x;

        yCoord = y;

    }

/\*\* *Создает новое местоположение с координатами (0, 0). \**\*/

*public* Location() {

        this(0, 0);

    }

*public* *boolean* equals(*Object* obj) {

        if (this == obj) {

            return true;

        }

        if (!(obj instanceof Location)) {

            return false;

        }

*Location* location = (Location) obj;

        return xCoord == location.*xCoord* && yCoord == location.*yCoord*;

    }

*public* *int* hashCode() {

        return Objects.hash(xCoord, yCoord);

    }

}

**Код файла** Location.java

/\*\*

*\* Этот класс представляет собой простую двумерную карту, состоящую из*

*\* квадратных ячеек.*

*\* В каждой ячейке указывается стоимость обхода этой ячейки.*

*\**\*/

*public* *class* Map2D {

/\*\* *Ширина карты. \**\*/

*private* *int* width;

/\*\* *Высота карты. \**\*/

*private* *int* height;

/\*\*

*\* Фактические картографические данные, необходимые алгоритму поиска пути для*

*\* навигации.*

*\**\*/

*private* *int*[][] cells;

/\*\* *Начальное местоположение для выполнения поиска пути A\*. \**\*/

*private* *Location* start;

/\*\* *Конечное местоположение для выполнения поиска пути A\*. \**\*/

*private* *Location* finish;

/\*\* *Создает новую 2D-карту с заданными шириной и высотой. \**\*/

*public* Map2D(*int* width, *int* height) {

        if (width <= 0 || height <= 0) {

            throw new *IllegalArgumentException*(

"width and height must be positive values; got "+ *width* +

"x"+ *height*);

        }

        this.*width* = width;

        this.*height* = height;

        cells = new *int*[width][height];

        // *Составьте некоторые координаты для начала и конца.*

        start = new *Location*(0, *height* /2);

        finish = new *Location*(*width* -1, *height* /2);

    }

/\*\*

*\* Этот вспомогательный метод проверяет указанные координаты, чтобы увидеть,*

*\* находятся ли они*

*\* в пределах границ карты. Если координаты не указаны на карте*

*\* затем метод выдает исключение <code>IllegalArgumentException</code>.*

*\**\*/

*private* *void* checkCoords(*int* x, *int* y) {

        if (x < 0 || x > width) {

            throw new *IllegalArgumentException*("x must be in range [0, "+

*width* +"), got "+ *x*);

        }

        if (y < 0 || y > height) {

            throw new *IllegalArgumentException*("y must be in range [0, "+

*height* +"), got "+ *y*);

        }

    }

/\*\* *Возвращает ширину карты. \**\*/

*public* *int* getWidth() {

        return width;

    }

/\*\* *Возвращает высоту карты. \**\*/

*public* *int* getHeight() {

        return height;

    }

/\*\*

*\* Возвращает значение true, если указанные координаты содержатся на карте*

*\* площадь.*

*\**\*/

*public* *boolean* contains(*int* x, *int* y) {

        return (x >= 0 && x < width && y >= 0 && y < height);

    }

/\*\*

*\* Возвращает значение true, если местоположение содержится в области карты.*

*\**\*/

*public* *boolean* contains(*Location* loc) {

        return *contains*(loc.*xCoord*,loc.*yCoord*);

    }

/\*\* *Возвращает сохраненное значение стоимости для указанной ячейки. \**\*/

*public* *int* getCellValue(*int* x, *int* y) {

*checkCoords*(*x*, *y*);

        return cells[x][y];

    }

/\*\* *Возвращает сохраненное значение стоимости для указанной ячейки. \**\*/

*public* *int* getCellValue(*Location* loc) {

        return *getCellValue*(loc.*xCoord*,loc.*yCoord*);

    }

/\*\* *Задает значение стоимости для указанной ячейки. \**\*/

*public* *void* setCellValue(*int* x, *int* y, *int* value) {

*checkCoords*(*x*, *y*);

        cells[x][y] = value;

    }

/\*\*

*\* Возвращает начальное местоположение для карты. Это где сгенерированный*

*\* путь начнется с.*

*\**\*/

*public* *Location* getStart() {

        return start;

    }

/\*\*

*\* Задает начальное местоположение для карты. Здесь находится сгенерированный*

*\* путь*

*\* начнется с.*

*\**\*/

*public* *void* setStart(*Location* loc) {

        if (loc == null)

            throw new *NullPointerException*("loc cannot be null");

        start = loc;

    }

/\*\*

*\* Возвращает конечное местоположение для карты. Это где сгенерированный*

*\* путь завершится.*

*\**\*/

*public* *Location* getFinish() {

        return finish;

    }

/\*\*

*\* Задает конечное местоположение для карты. Здесь находится сгенерированный*

*\* путь*

*\* завершится.*

*\**\*/

*public* *void* setFinish(*Location* loc) {

        if (loc == null)

            throw new *NullPointerException*("loc cannot be null");

        finish = loc;

    }

}

**Код файла** Map2D.java

/\*\*

*\* Этот класс представляет собой один шаг в пути, сгенерированном A\* pathfinding*

*\* алгоритм. Путевые точки состоят из местоположения, предыдущей путевой точки в*

*\* пути и некоторых значений затрат, используемых для определения наилучшего*

*\* пути.*

*\**\*/

*public* *class* Waypoint {

/\*\* *Местоположение этой путевой точки. \**\*/

*Location* loc;

/\*\*

*\* Предыдущая путевая точка на этом пути или <код>null</code>, если это*

*\* корень поиска A\*.*

*\**\*/

*Waypoint* prevWaypoint;

/\*\*

*\* В этом поле хранится общая предыдущая стоимость получения от начальной*

*\* местоположение до этой путевой точки, через цепочку путевых точек. Это*

*\* фактическая стоимость следования по пути; она не включает никаких оценок.*

*\**\*/

*private* *float* prevCost;

/\*\*

*\* В этом поле хранится оценка оставшейся стоимости поездки из*

*\* эта путевая точка к конечному пункту назначения.*

*\**\*/

*private* *float* remainingCost;

/\*\*

*\* Создайте новую путевую точку для указанного местоположения. Предыдущая*

*\* путевая точка*

*\* может быть дополнительно указано, или ссылка может быть <code>null</code>,*

*\* чтобы*

*\* указать, что путевая точка является началом пути.*

*\**\*/

*public* Waypoint(*Location* loc, *Waypoint* prevWaypoint) {

        this.*loc* = loc;

        this.*prevWaypoint* = prevWaypoint;

    }

/\*\* *Возвращает местоположение путевой точки. \**\*/

*public* *Location* getLocation() {

        return loc;

    }

/\*\*

*\* Возвращает предыдущую путевую точку в пути или <код>null</code>, если это*

*\* - это начало пути.*

*\**\*/

*public* *Waypoint* getPrevious() {

        return prevWaypoint;

    }

/\*\*

*\* Этот мутатор позволяет устанавливать как предыдущую стоимость, так и*

*\* оставшуюся стоимость*

*\* в одном вызове метода. Обычно эти значения будут установлены на одном и том*

*\* же*

*\* время в любом случае.*

*\**\*/

*public* *void* setCosts(*float* prevCost, *float* remainingCost) {

        this.*prevCost* = prevCost;

        this.*remainingCost* = remainingCost;

    }

/\*\*

*\* Возвращает фактическую стоимость перехода к этой точке с начальной*

*\* местоположение, через ряд путевых точек в этой цепочке.*

*\**\*/

*public* *float* getPreviousCost() {

        return prevCost;

    }

/\*\*

*\* Возвращает оценку оставшейся стоимости поездки из этого*

*\* укажите на конечный пункт назначения.*

*\**\*/

*public* *float* getRemainingCost() {

        return remainingCost;

    }

/\*\*

*\* Возвращает общую смету затрат для этой путевой точки. Это включает в*

*\* себя \* фактическую стоимость проезда до этой точки из исходного*

*\* местоположения, плюс*

*\* оценка оставшейся стоимости поездки из этой точки до*

*\* конечного пункта назначения.*

*\**\*/

*public* *float* getTotalCost() {

        return prevCost + remainingCost;

    }

}

Код файла Waypoint.java

**Вывод**

В ходе выполненной работы я ознакомился с методом AStar. Смог адаптировать алгоритм под нашу задачу.

**Список источников**

1. Камаев В.А., Костерин В.В. Технологии программирования. М.: Высшая школа, 2006.

2. Жоголев Е.А.Технология программирования. – М.: Научный мир, 2004.