

В этом документе будет рассказано о проделанной работе, даны пояснения к используемым алгоритмам.

Изучая курс по комбинаторике и теории графов, мы были в восторге от алгоритмов поиска, и очень хотелось как-нибудь проверить их на деле, так сказать столкнуть их лицом к лицу и реализовать их все. Ничего лучше не придумали как создать огромную карту, “построить” на ней города, заселить её ботами, которые будут ходить из одной точки в другую, используя разные алгоритмы поиска. Получается как бы игра, где игрок не участвует, такой жанр называется ZPG. Боты будут “жить” в этом огромном сгенерированном мире сами, выполняя случайные задания, которые будут получать в случайно расположенных городах, и получая за выполнения этих заданий очки. А содержание этих заданий примерно такое: “Посети следующие координаты...” Понятное дело, что бот, который использует “лучший” алгоритм поиска будет иметь наибольшее количество очков на длинной дистанции. Таким образом можно найти этот самый лучший алгоритм поиска. Обо всём по порядку.

Создание карты и городов

Одна из самых первых проблем с которой необходимо столкнуться - это создание карты. Большой карты. Проблем несколько:

- 1) Как вообще генерировать гигантскую карту?
- 2) Сколько эта карта будет весить?

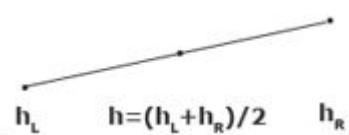
Для генерации карты было решено использовать алгоритм Diamond-Square. Суть примерно в следующем:

Карта - это двумерный массив размером $(2^n + 1)$ на $(2^n + 1)$. Элементы этого массива - это высота в данной точке. Рассмотрим на примере, например для 2D:

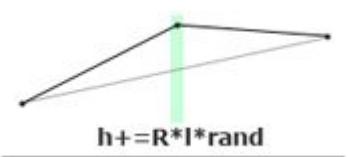
Взять 2 точки:



Найти среднее:



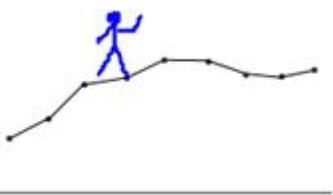
Немножко поправить:



И уйти в рекурсию...

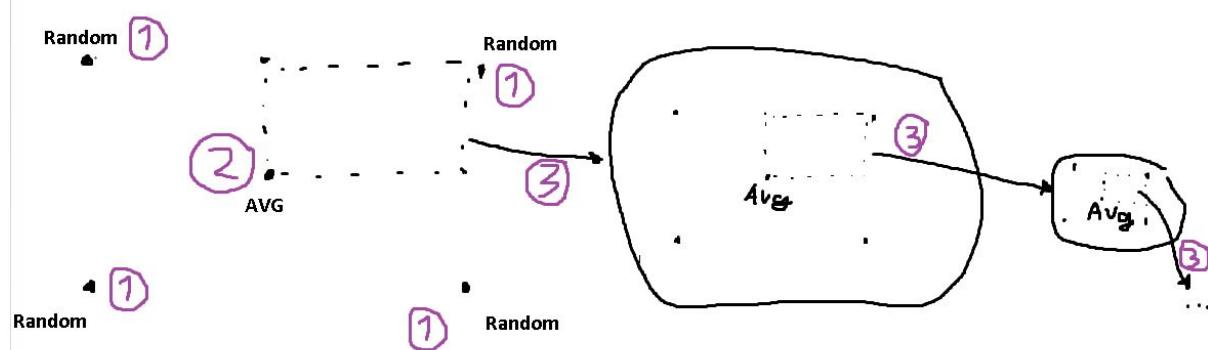


Получилась карта, теперь можно даже поставить человечка:



Но это 2D, что же делать с 3D?

Принцип почти такой же. 1) Надо выбрать четыре точки. 2) Найти в центре среднее арифметическое + поправка и 3) уйти благополучно в рекурсию...



Но в итоге получается какая-то ерунда. Всё исправляет **Алгоритм diamond-square**:

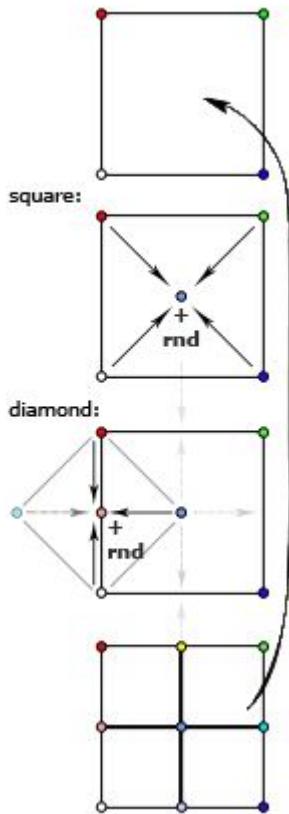
Суть проста: просто ещё добавляются к прямоугольникам (square) ромбики (diamond):

Начинается работа с двумерного массива размера $2^n + 1$. Элементы этого массива - это высота в данной точке.

В четырех угловых точках массива устанавливаются начальные значения высот. Шаги diamond и square выполняются поочередно до тех пор, пока все значения массива не будут установлены.

Шаг diamond (ромб). Для каждого ромба в массиве, устанавливается срединная точка, которой присваивается среднее арифметическое из четырех угловых точек плюс случайное значение (будем называть его **дефектом**).

Шаг square (квадрат). Для каждого квадрата в массиве, находится срединная точка, в которую устанавливается среднее значение четырех угловых точек плюс случайное значение (**дефект**).

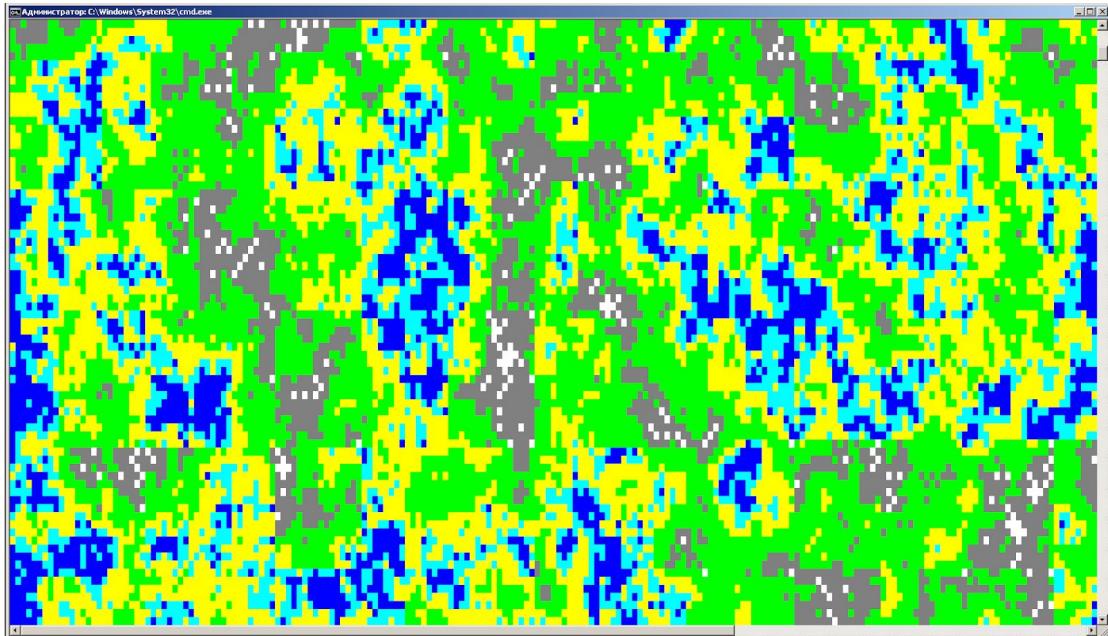


Теперь получается карта, где в каждой точке определена высота. Ну например 0 - это вода, 1 - это земля, 2 - это холм, 3 - это гора, 4 - это пик... И так далее. Но что, если сгенерировать 2 такие карты, где первая будет картой высот, а вторая картой, например, влажности? Получится в каждой точке одновременно задана и высота, и влажность (ну или ещё что-нибудь). Получается генерация "биомов" (да, как в майнкрафте). Это может выглядеть следующим образом:

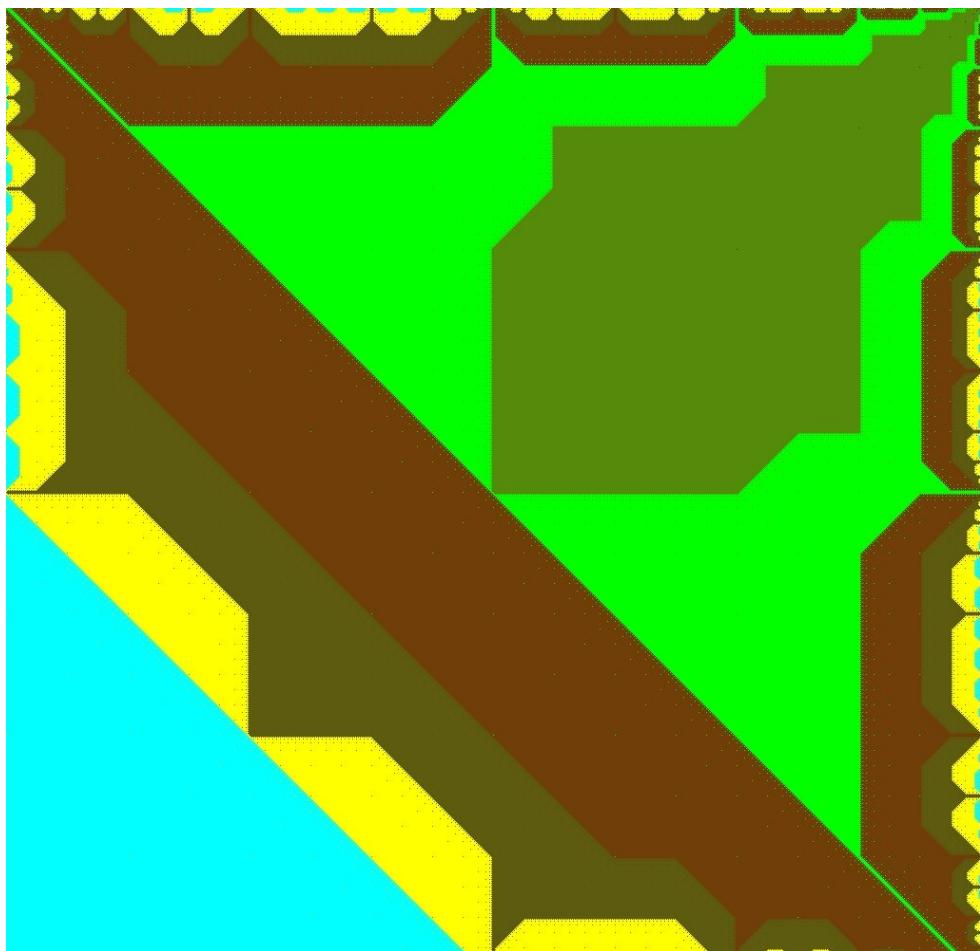


В некотором смысле можно сказать, что это алгоритм генерирования произвольной (с некоторыми ограничениями) непрерывной функции двух переменных. Его можно использовать не только для генерирования карты.

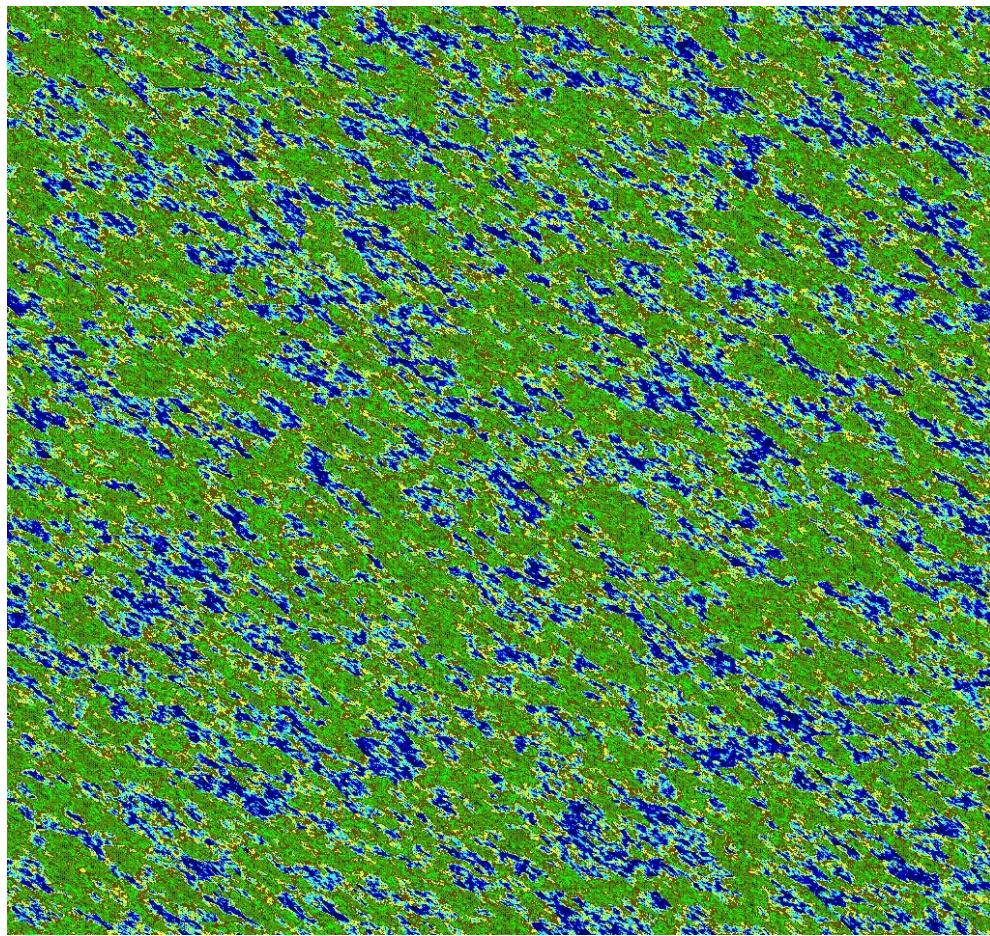
Очень многое зависит от выбора этого самого случайного значения (дефекта), которое прибавляется на каждом шаге. Плюс ко всему нужно правильно подобрать “блоки” для каждого значения высоты и влажности. Вот как изменилась карта в зависимости от изменения этих значений:

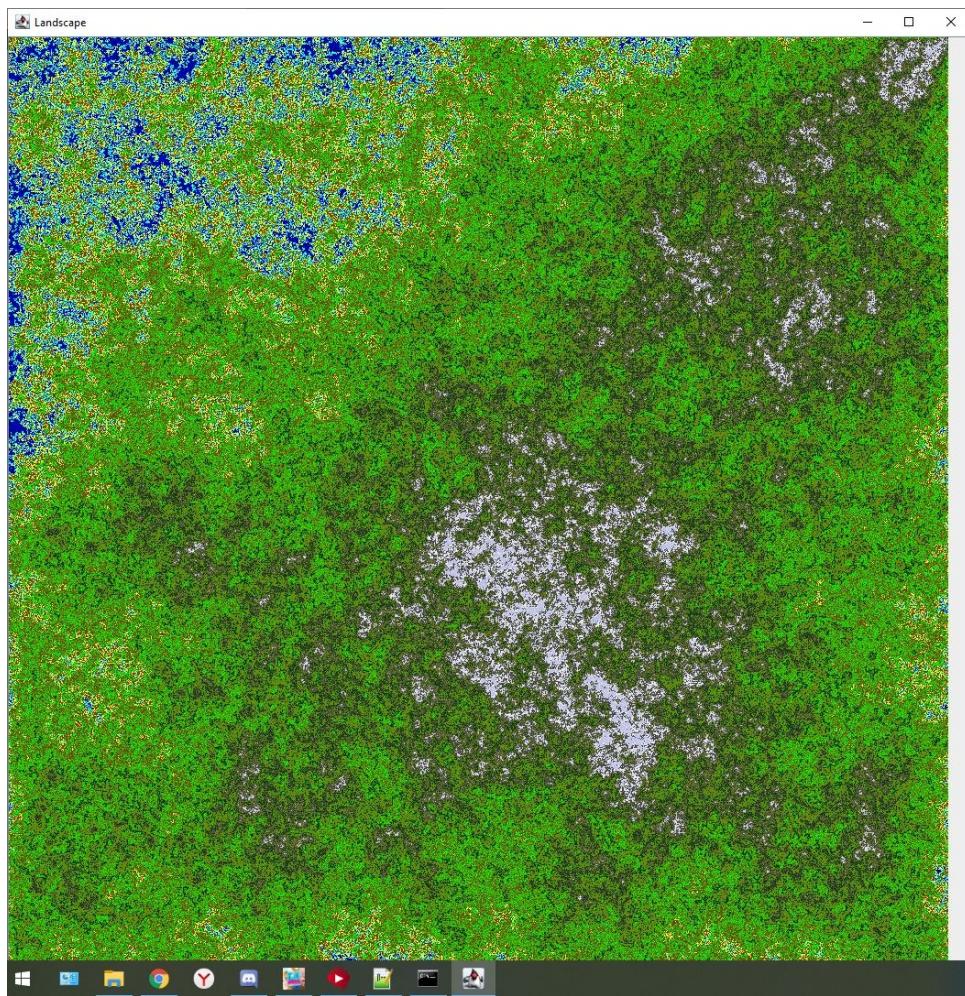


После перешли с консоли на нормальное GUI. Вот что будет если совсем пренебречь этим рандомным значением (дефектом):

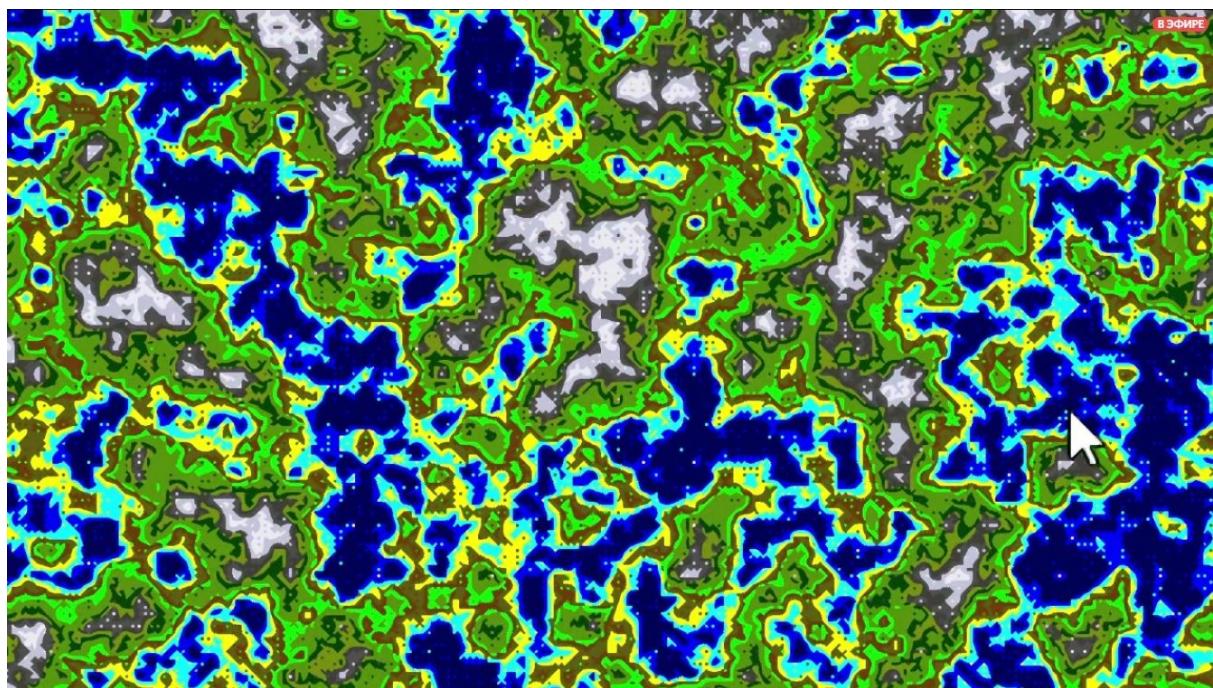


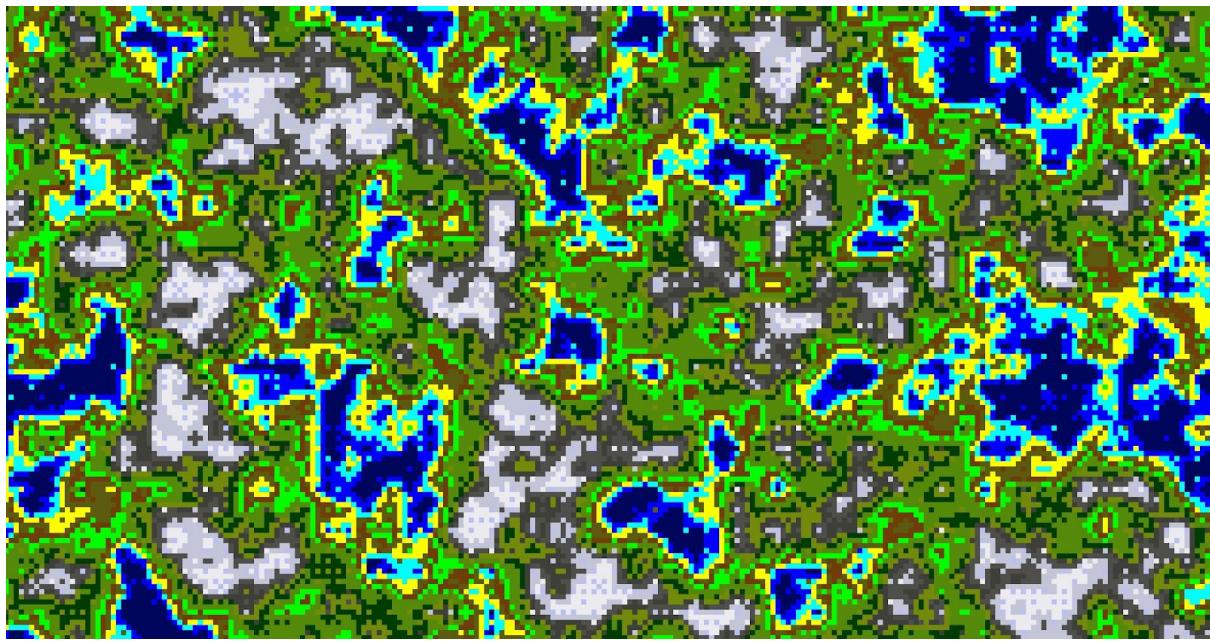
Играясь с разными значениями дефекта, можно получать совсем разные местности (масштаб одинаковый):





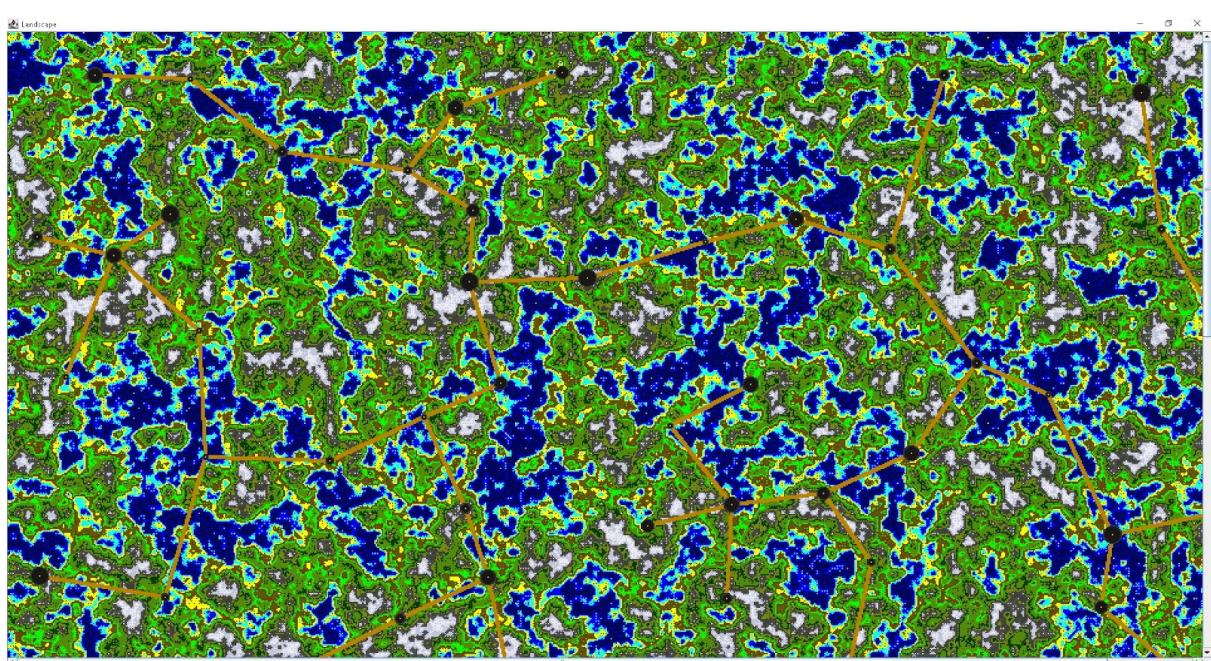
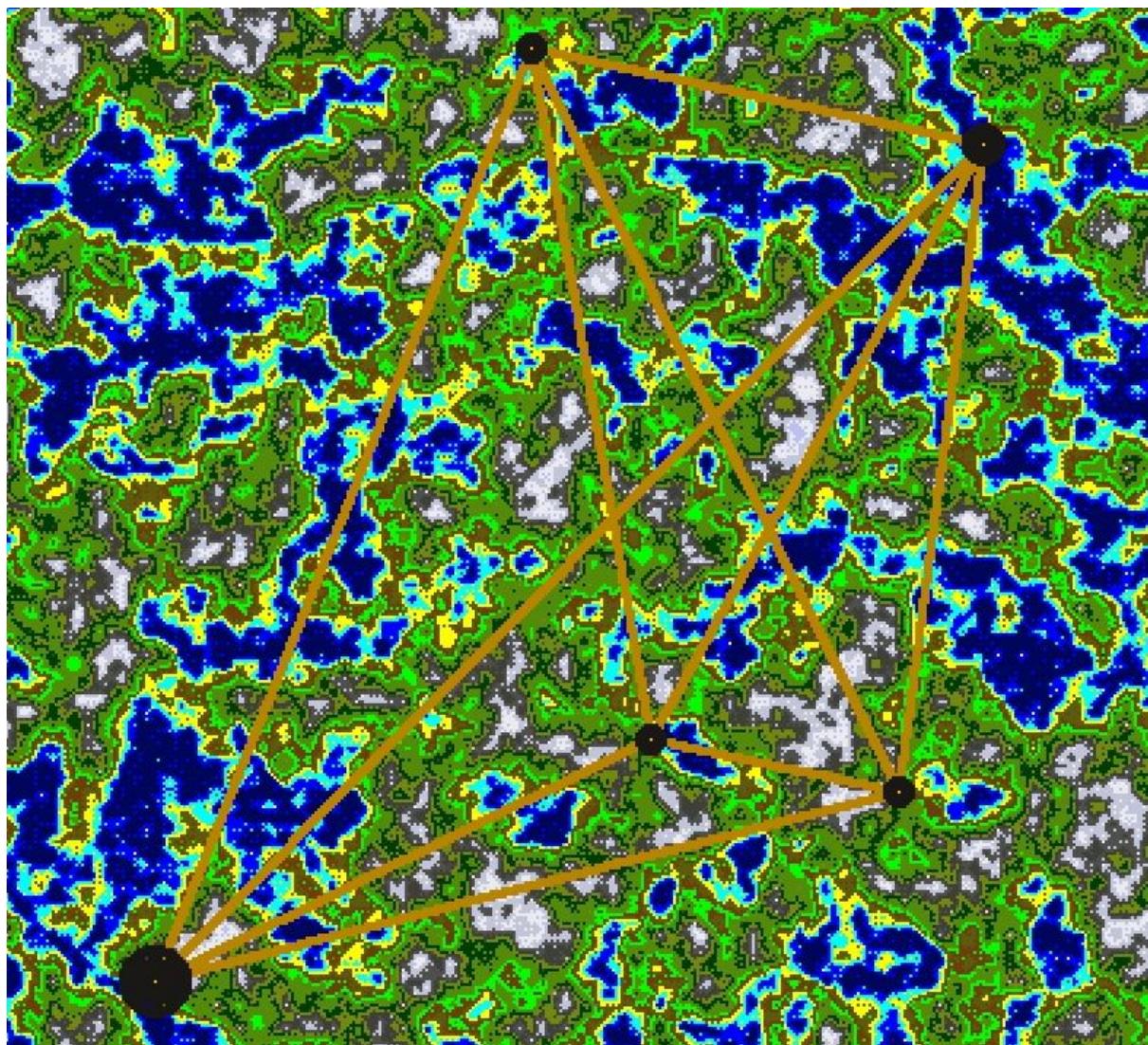
Тут мы решили сглаживать кубики, но это очень сильно лагало)

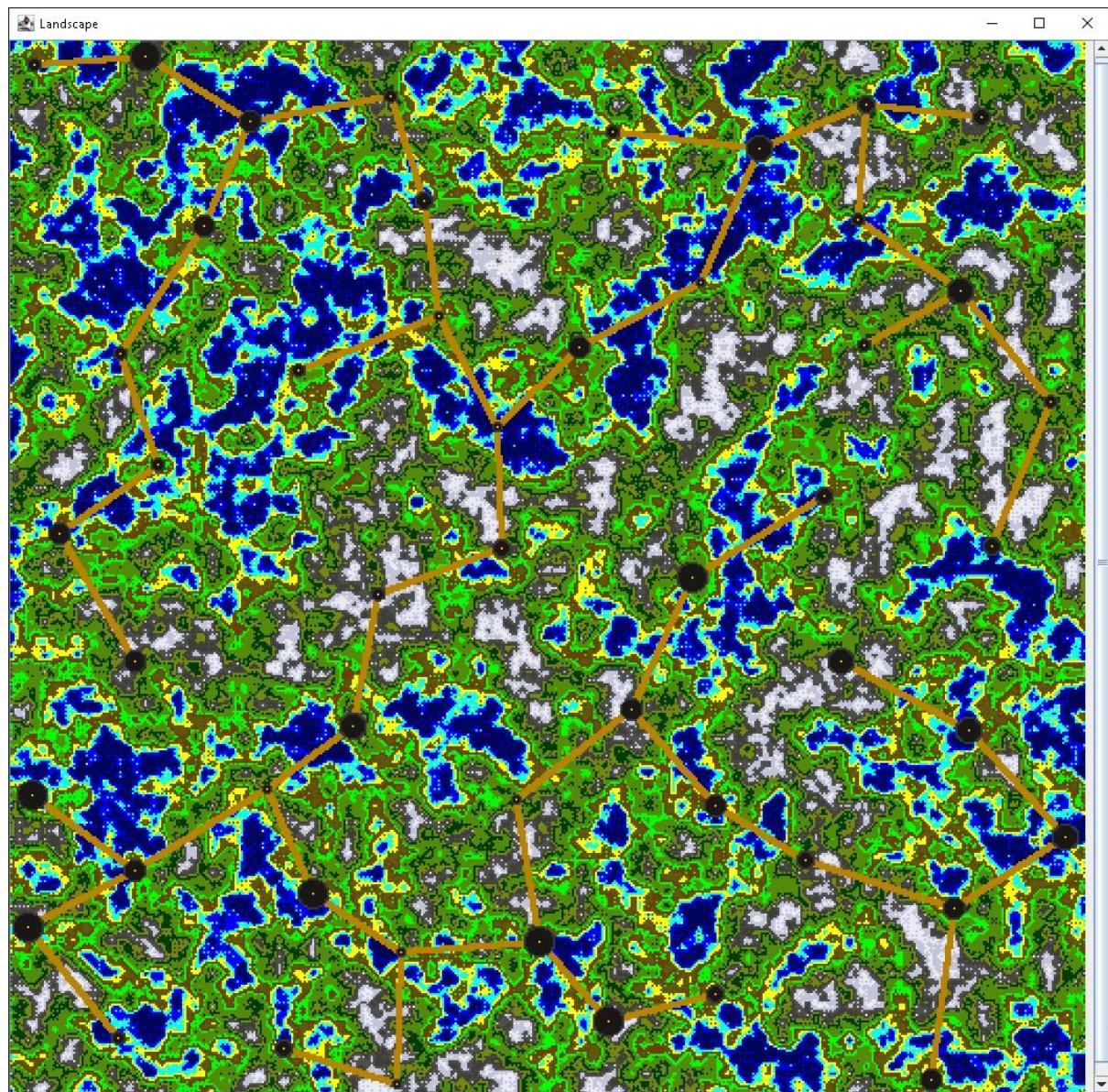


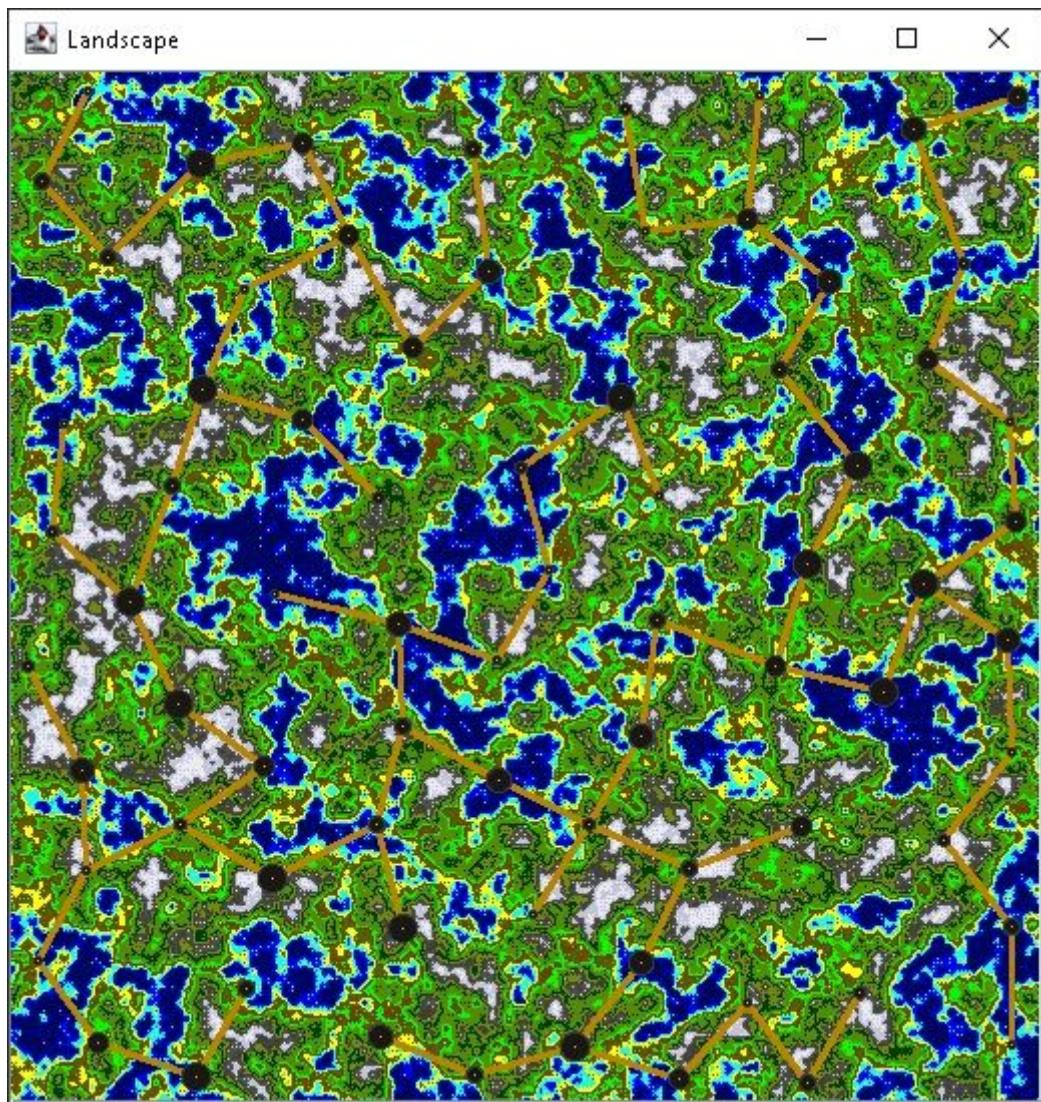


Добавление городов

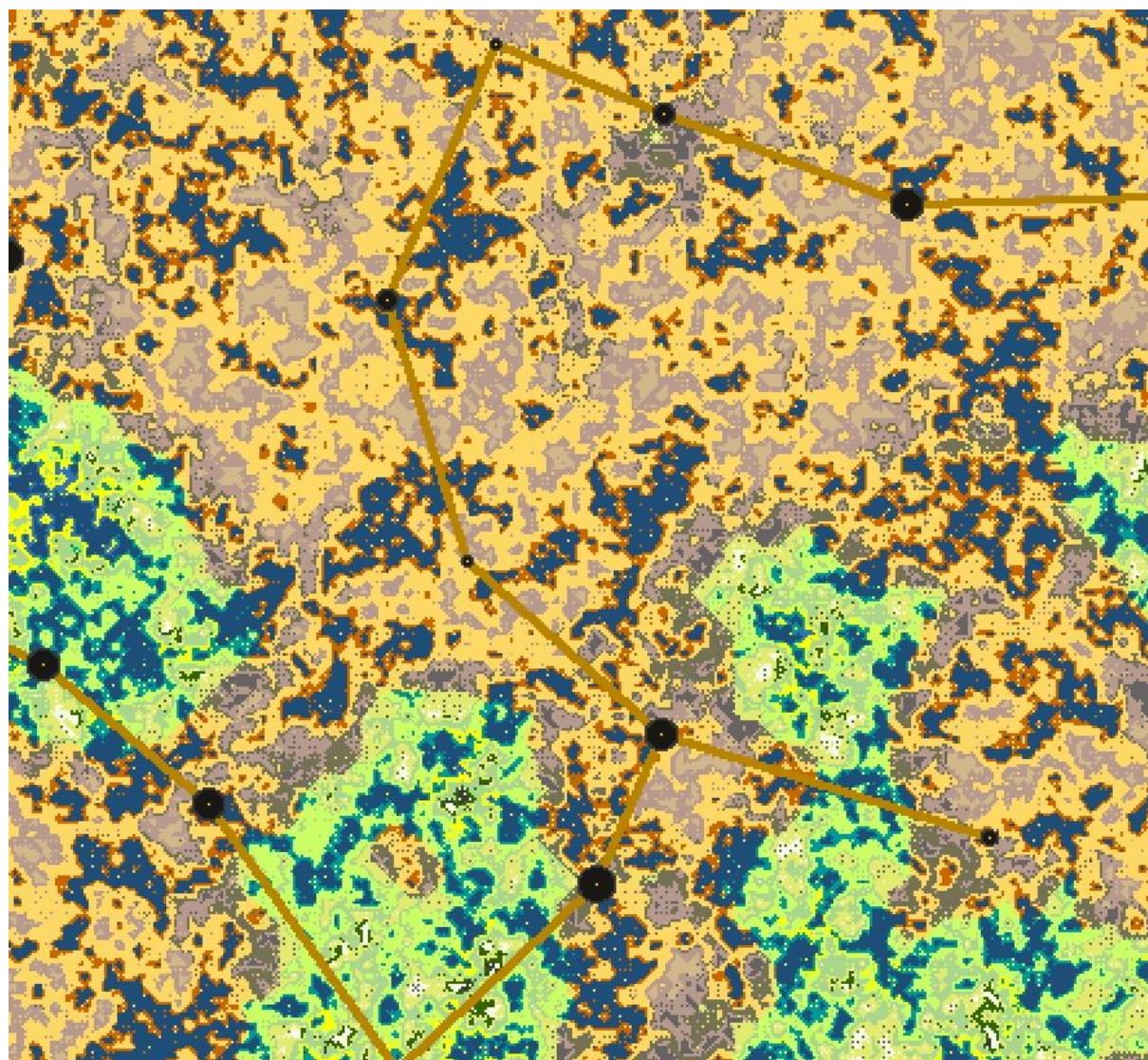
Тут мы просто расставляли города также, как бы расставляли “руками”. Просто тыкаем город в рандомную точку (при этом смотрим, чтобы не было слишком близко другого города и не вылезало за карту). Дальше самое интересное: расставить дороги. Тут просто можно использовать алгоритм Прима, где вес ребер - это расстояние между городами, а уже потом можно добавить ещё дороги, если уж так сильно нужно:

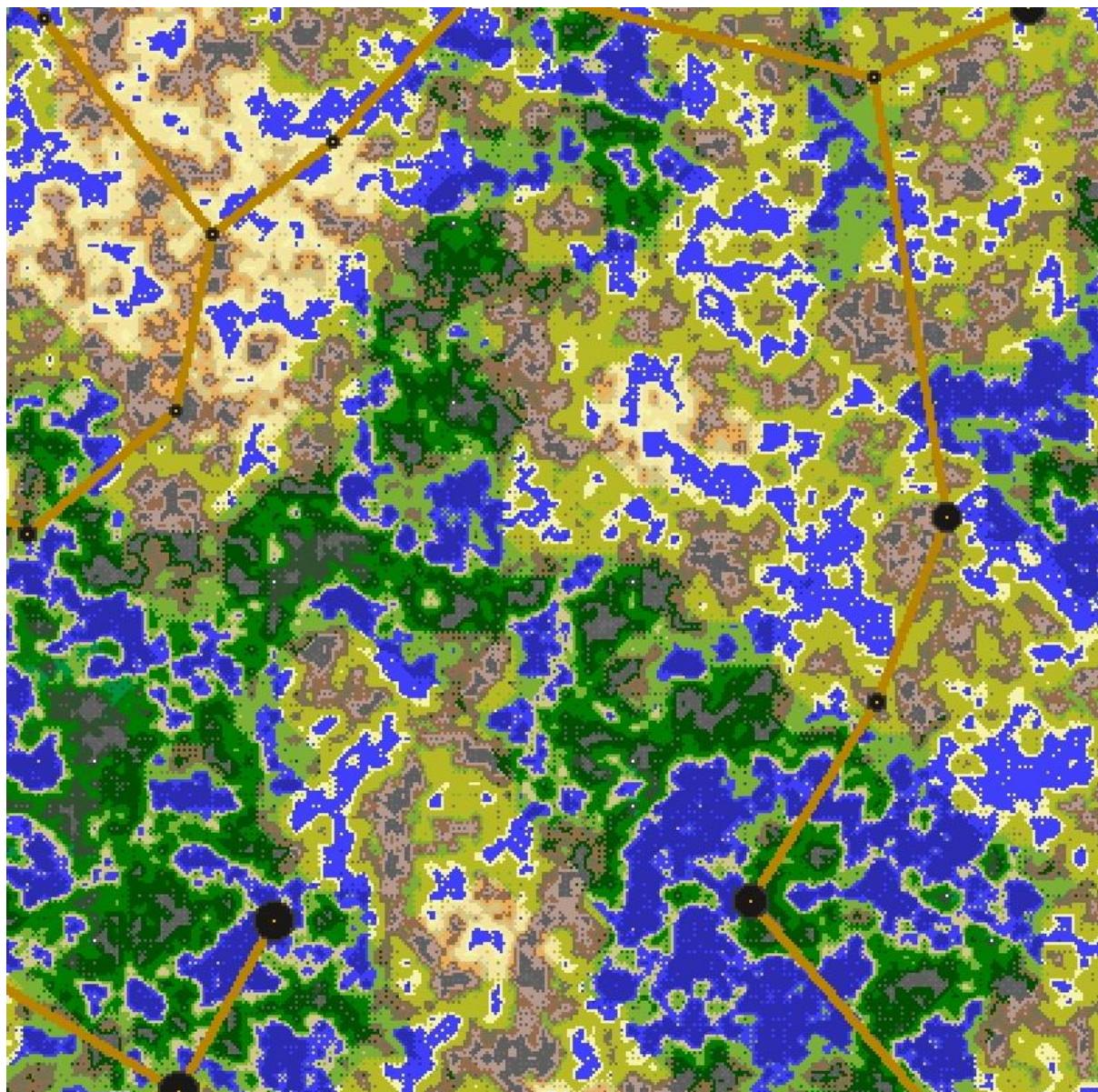






Остается правильно подобрать “блоки” при заданном значении высоты и влажности и стоимость перехода между блоками:





Мы остановились на таком выборе:

Влажность(0=сухо -> 99=влажно)										
Высота (0=низко -> 99=высоко)	9	8	7	6	5	4	3	2	1	0
	50					40				
					30					
				15		12		22		
							13			
							19		15	
									16	
								11	18	
										5
										25

Циферки обозначают сложность ходьбы по данному блоку (по дороге - это единица), а стоимость перехода между блоками мы решили взять за среднее арифметическое сложности ходьбы по блокам, между которыми бот идёт. Сложность будет выражаться во времени, которое

тратит бот, чтобы перейти с одного блока на другой. Например, сложность 20 означает, что бот будет двигаться 20 “тиков” (или единиц времени) к следующему блоку. Для дороги сложность равна единице.

Саму карту взяли 1025 на 1025 блоков (т.е. больше **миллиона**). Получается граф, где вершины - это “блоки”, а ребра есть только к соседним блокам (по диагонали не считается), причём определен их вес.

Сколько же может весить данная карта? Есть 2 варианта, в карте, представляющей собой массив, в каждой ячейке можно хранить ссылку на объект Блок, который будет хранить какую-нибудь информацию о себе и так далее. То есть ссылка 8 байт + информация блока о себе где-то 15 байт. Всего таких блоков $1025 \times 1025 = 1050625$ шт. Тогда примерный вес карты может быть: $1050625 * (8+15) = 24164375$ байт = 23 Мегабайт. Или же можно заранее подготовить все блоки, и ссылаться на них. Всего нужно подготовить блоков $10 \times 10 = 100$ штук. Тогда вес может быть примерно таким: $1050625 * 8 + 100 * 15 = 8406500$ байт = 8 Мегабайт. Как окажется дальше это совсем немного по сравнению с памятью, которую будут использовать алгоритмы поиска.

Вспомогательные структуры данных

Во всех алгоритмах поиска в дальнейшем будет нужно множество. Постоянно проверяется есть ли блок в каком-то множестве или нет? Например, проверяется посещена ли уже вершина? Как мы писали выше в нашем графе свыше миллиона блоков. И проверять каждый раз есть ли вершина в каком-то множестве, где уже 500 тысяч вершин КАЖДУЮ итерацию ооочень долго. На самом деле это действительно проблема. Мы её решили. Придумали такую структуру данных, где проверка на вхождение в множество, добавление в множество и удаление из множества выполняется за $O(1)$. Даже когда в множестве 1000000 вершин. Правда здесь нужно оговориться. Чтобы это выполнялось, нужно каждому элементу этого множества поставить в соответствие число, причем это должна быть биекция... То есть каждому числу должно соответствовать одна вершина и наоборот. В нашем случае это выполняется, ведь вершина - это блок, у блока есть координата (x, y) и размер карты известен = 1025×1025 , поэтому этим числом будет $x+y*1025$. Как бы все точки выстраиваются в одну строчку. Но и размер такого множества для 1025×1025 элементов будет около 32 Килобайт. Причём всегда и не важно сколько там уже содержится точек, за всё нужно платить. Суть в следующем:

В “инте” 4×8 бит, каждому биту можно сопоставить вхождение вершины в множестве.

Создаём массив размером $((1025 \times 1025) / (4 \times 8) + 1)$, заполняем его нулями. Всё, пустое множество готово.

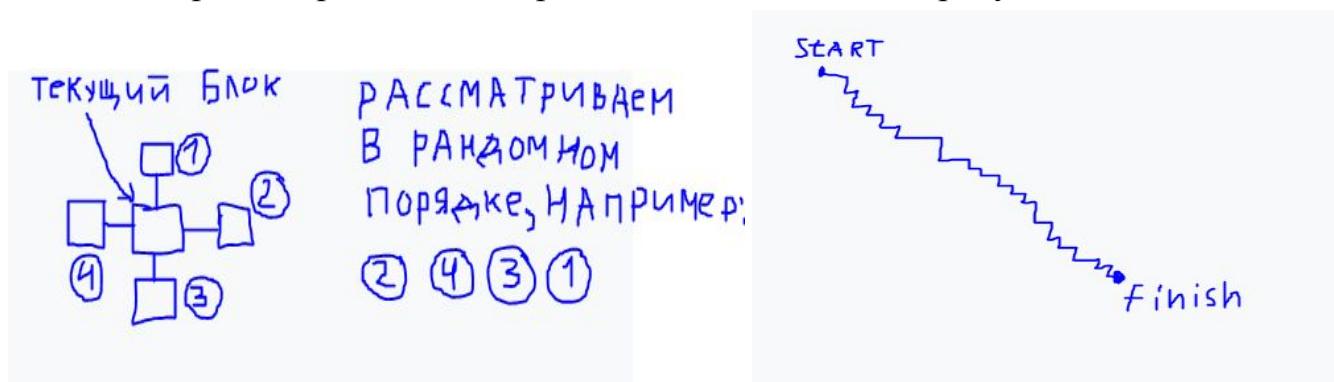
Теперь нужно просто найти нужный бит для элемента с номером K. Пусть этот массив - это массив a, тогда нужный бит: $a[K / (4 \times 8)] \& (1 \ll K \% (4 \times 8))$.

Алгоритмы поиска

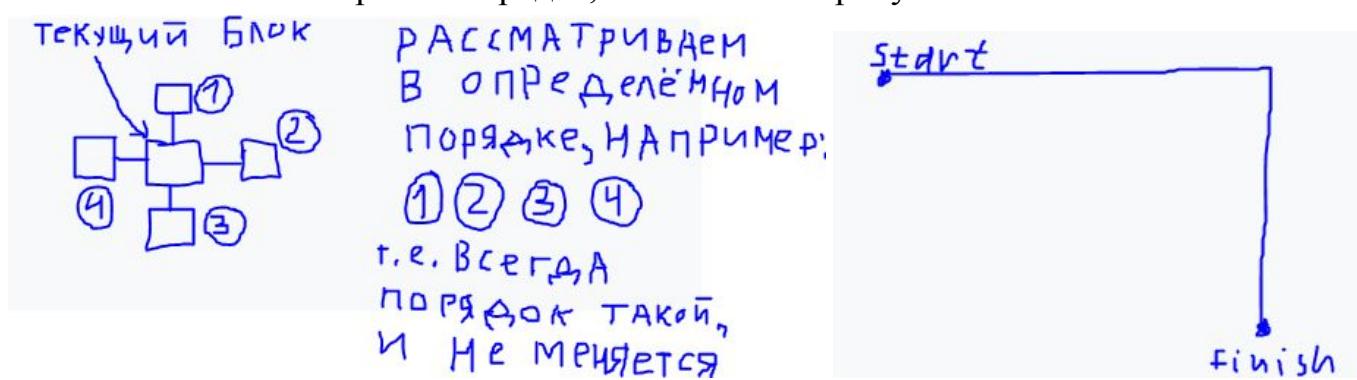
Алгоритмы поиска в ширину и глубину

Это самые стандартные алгоритмы поиска в графах. Они быстро ищут путь, но сами боты передвигаются долго.

Алгоритм поиска в ширину (Breadth First Search) даёт такой путь, чтобы совершить как можно меньше переходов с одного блока на другой, не рассматривая стоимость. Причём, путь будет “диагональным”, если рассматривать блоки рандомно, то есть как на рисунке:

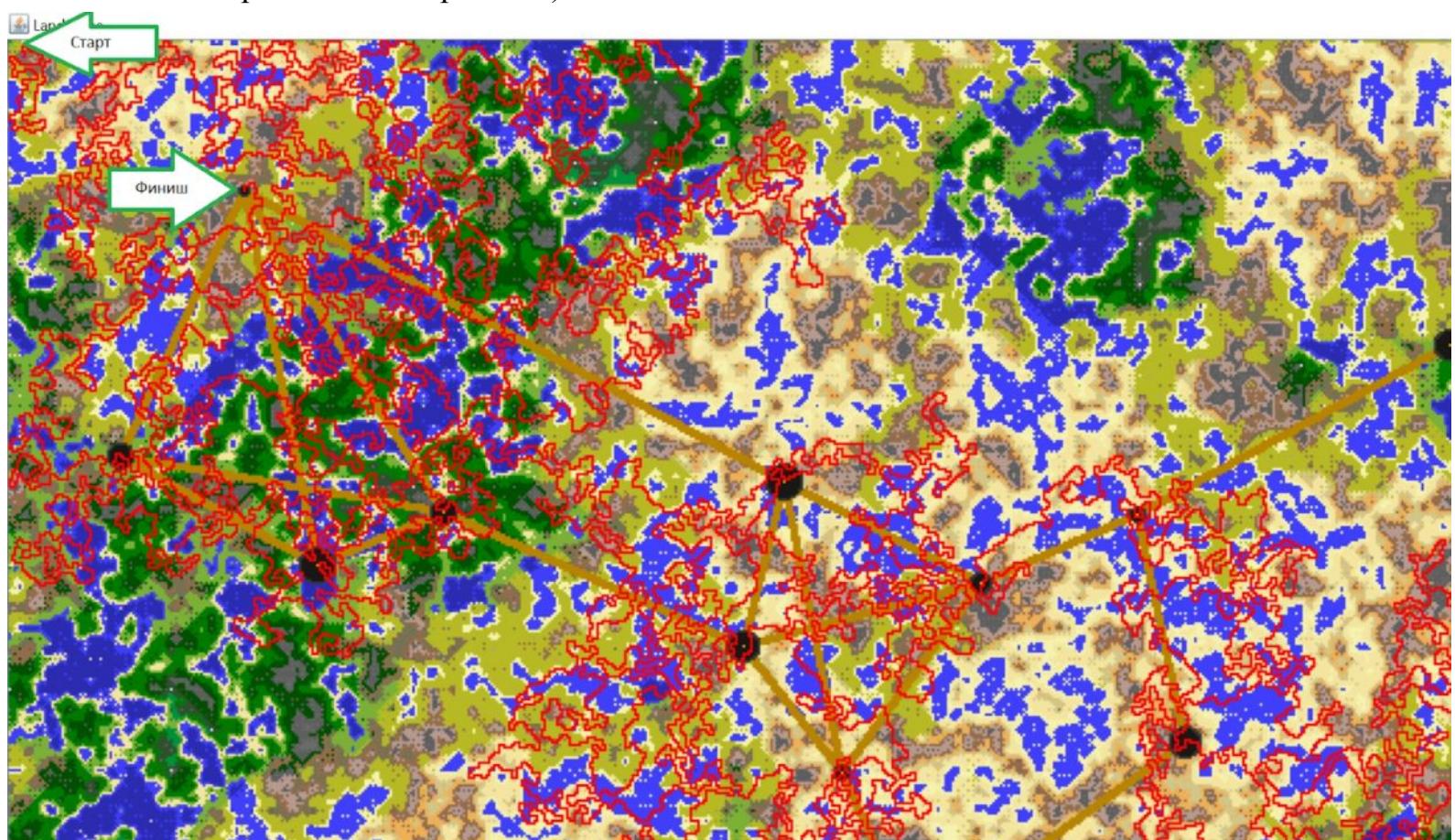


Или путь может быть угловым, если рассматривать рядом стоящие блоки в строгом порядке, то есть как на рисунке:

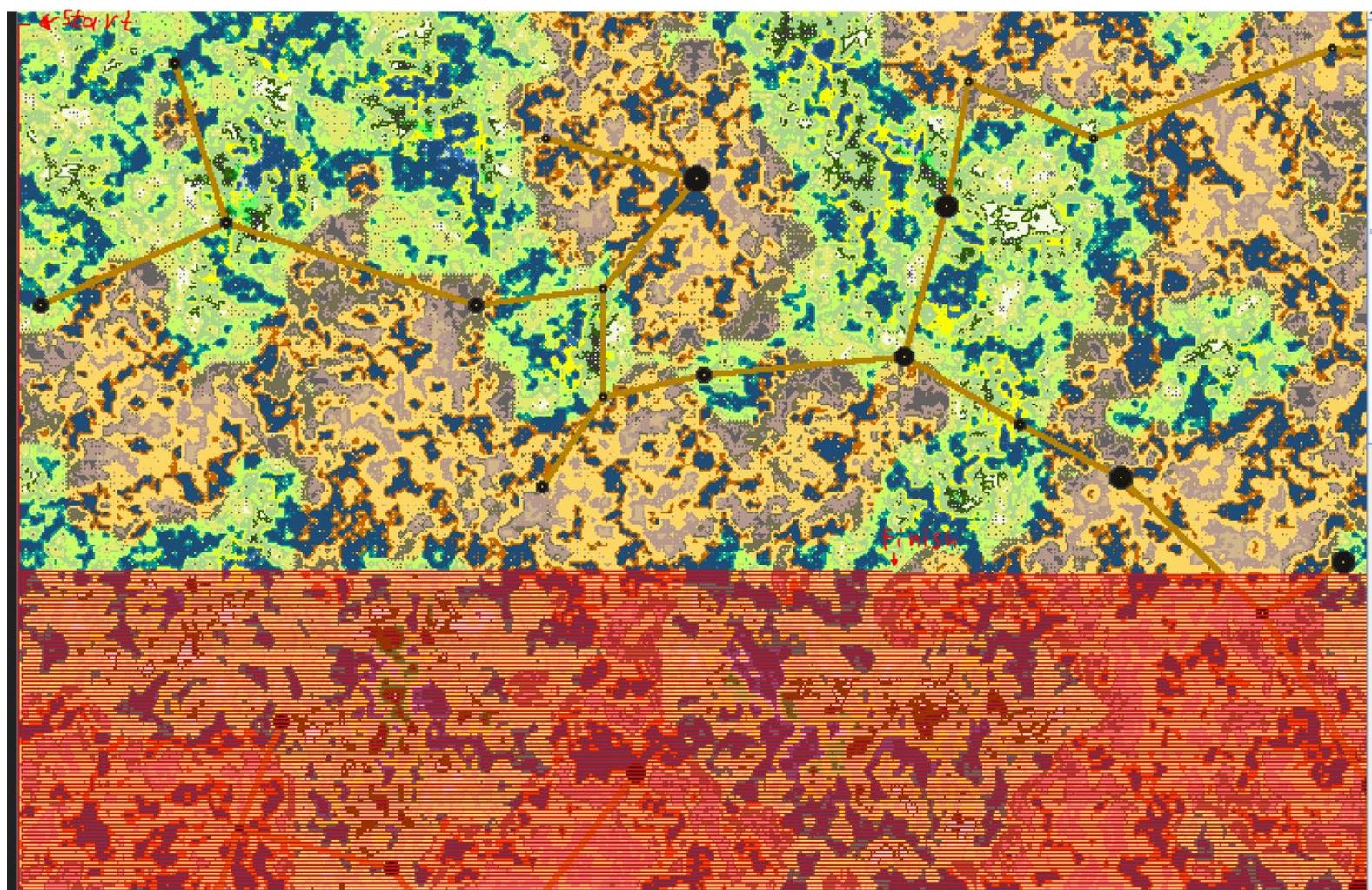


Очень интересные [результаты](#) показывает алгоритм поиска в глубину (Depth First Search). Опять такие результаты будут разные в зависимости от порядка, в котором рассматриваются блоки, но и там, и

там вместо того, чтобы пройти как можно короче (хотя бы в каком-нибудь смысле), алгоритм поиска в глубину устраивает “экскурсию” по всей карте. Примерно следующим образом (красным изображён сам путь, построенный алгоритмом):



На рисунке сверху изображен путь, если рассматривать блоки рандомно. Если же из перебирать строго определенном порядке, то будет примерно такое:



Волновой алгоритм поиска (алгоритм Ли)

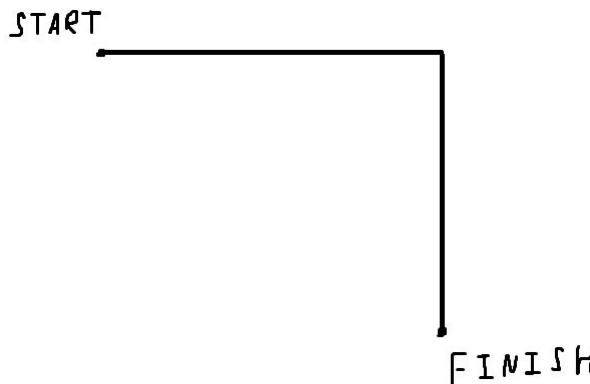
Так как есть 2 варианта данного алгоритма: для взвешенного и не взвешенного графа, то мы решили реализовать оба и посмотреть, кто из них будет лучше и при каких условиях.

Угловой алгоритм

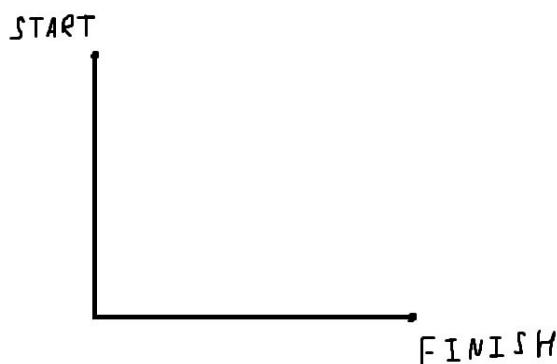
В некотором плане, угловой алгоритм - частный случай поиска в ширину, когда соседи вершины рассматриваются в определенном порядке (пример можно посмотреть [тут](#)). В связи с этим пришла идея реализации данного алгоритма следующим образом: случайно выбираем ось, по которой мы будем идти вначале до момента выравнивания с финишем, т.е. до того, когда текущая позиция будет лежать на одной прямой с конечной точкой.

Итого получаем 2 возможных варианта пути:

- Выравниваемся по оси ОХ, далее идем по прямой к финишу:



- Выравниваемся по оси OY, далее идем по прямой к финишу:



Диагональный алгоритм

Диагональный алгоритм - наследник углового алгоритма с одним маленьkim, но значимым отличием - после каждого шага меняется ось, по которой мы движемся. Если на нынешнем шаге мы шагнули по оси OX, и конечная вершина, к которой мы стремимся, не лежит на одной прямой с нашим местоположением, то на следующем шаге мы уже шагаем по оси OY.

Общая деталь диагонального и углового алгоритмов - мы выбираем самую приближенную вершину к концу (имеется в виду, что берем вершину, от которой нужно будет пройти наименьшее количество блоков). Таким образом отпадает необходимость помнить, какие вершины мы уже рассмотрели.

Алгоритм Дейкстры

Алгоритм Дейкстры в программе - пример того, как зависит результативность от реализации алгоритма. Сделав самый первый вариант алгоритма (у нас их таких 3), мы пришли к взаимному мнению, что нужно его улучшать. Тут ветка событий ушла в 2 стороны:

- Улучшение текущей реализации

- Попытка реализовать алгоритм совсем иначе

Столкнув эти 3 алгоритма вместе, получилось следующее:

- Улучшенный изначальный алгоритм позволил добиться улучшение результата в некоторых случаях вплоть до 15%
- Попытка реализации иначе, к сожалению, привела к неутешительному результату. Несмотря на хорошую теоретическую оценку, результативность снизилась в разы.

Все три реализации алгоритма находятся в репозитории с программой в папке ZPG/GameLogic/Searchers.

Названия реализаций:

- Оригинальный алгоритм - Dijkstra
- Улучшенный алгоритм - newDijkstra
- Иной подход к реализации - DijkstraSearcher

Алгоритм Беллмана-Форда

Алгоритм, который показал себя, бесспорно, лучше всех. Для его реализации мы использовали следующий алгоритм:

У нас имеется очередь вершин к рассмотрению и список расстояний от старта до некоторой вершины на карте

ПОКА очередь не пуста И поиск пути не закончен

берем вершину из очереди

ЕСЛИ эта вершина - пункт назначения ТО завершаем поиск
ИНАЧЕ рассматриваем всех соседей текущей вершины

ЕСЛИ путь через текущую вершину будет быстрее, чем путь, записанный у соседа на данный момент ТО обновляем его

ЕСЛИ “обновленный” сосед рассматривался нами когда-либо ранее ТО добавляем его в самое **начало** очереди

ИНАЧЕ добавляем его в **конец** очереди.

КОНЕЦ ПОКА

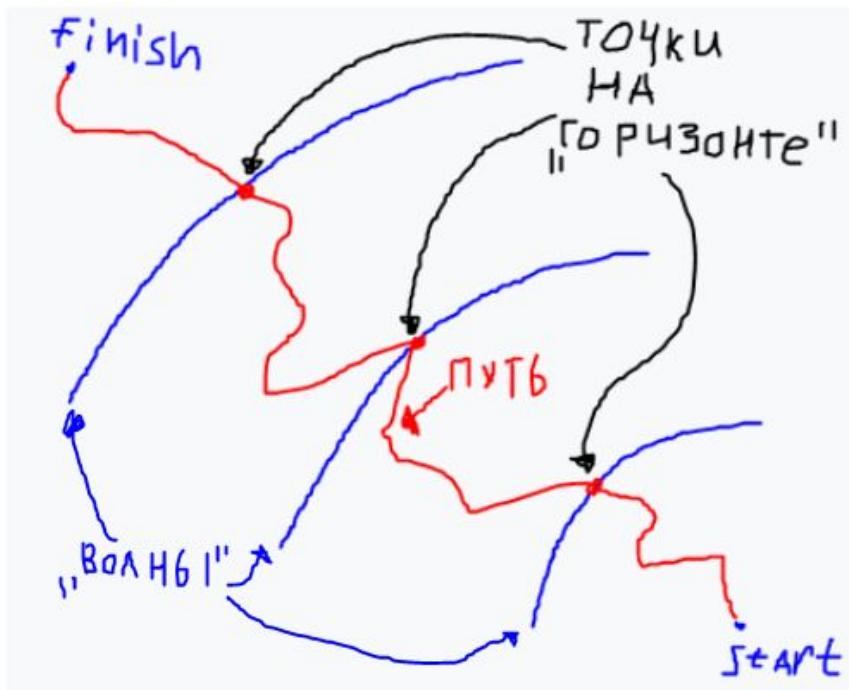
Алгоритм A*

Реализация вышла аналогичной алгоритму Дейкстры за исключением того, что отбирать вершины мы будем уже не по

минимальному расстоянию от старта, а по минимальному значению функции, состоящей из расстояния (имеется в виду проходимость) от старта до точки и расстояния (в блоках) до финиша.

Гибриды: алгоритмы Leekstra, LeeStar и LeeBellFord

Представим, что мы сами находимся на какой-то местности. Всю карту мы не видим, но зато можем смотреть в горизонт и выстраивать путь к точке на горизонте каким-нибудь алгоритмом (например, Дейкстрой, A* или Беллмана-Форда). Дойдя до неё, мы снова смотрим в горизонт и ищем путь до него. Получаются волны:



В названии какого алгоритма есть слово волновой? В алгоритме Ли, поэтому и придумали такое странное название:



Взяв за основу идею алгоритма Ли в плане, что мы движемся “волной”, мы скомпоновали алгоритмы Дейкстры, A* и Беллмана-Форда с данной идеей. Ограничиваая отбор вершин некоторым радиусом (расстояние от текущей точки до следующей точки на горизонте), мы находим наиболее оптимальную из них (наиболее приближенную к

финишу) и с помощью данных трех алгоритмов добираемся до нее. Таким образом, мы движемся несколько раз, пока не достигнем точки назначения.

Гибриды созданы для ускорения построения маршрутов на дальние дистанции. Согласно тестам, они со своей задачей справляются, несмотря на некоторый дефект в виде ответвления от маршрута:



Жизненный цикл ботов

Вначале каждого бота размещает случайным образом на карте. После каждый бот каждый “тик” делает следующее:

Если у него нет маршрута и нет задания, то нужно найти путь до ближайшего города и взять там задание.

Иначе следовать заданию

Результаты замеров

Карты для каждого замера - разные

худший результат	средний результат	лучший результат
------------------	-------------------	------------------

[Таблица с результатами](#)

Анализ замеров

Обратимся к [таблице](#) результатов, а именно - когда квесты выдавались случайно (вкладка AllQuests).

Название алгоритма	1	2	3	Место	вклад	
Дейкстра	2219	2670	828	10	2,78%	206006
Волновой (взвешенный граф)	156	74	495	11	0,35%	
Волновой (не взвешенный граф)	4651	7356	3236	8	7,40%	
Поиск в ширину	5311	7475	3704	7	8,00%	
A-стар	8843	8292	2971	4	9,76%	
Диагональный	5813	6593	4841	6	8,37%	
Угловой	5794	7250	4497	5	8,51%	
Ликстра	1123	3757	1520	9	3,11%	
Листар	8606	10214	5887	3	11,99%	
Беллмана-Форда	16202	18758	5503	2	19,64%	
ЛиБеллфорд	16577	18202	6588	1	20,08%	
МИН находится в N строке		3	3	3		
МАКС находится в N строке		12	11	12		
Ликстра лучше Дейкстры	Нет	Да	Да			
Листар лучше A*	Нет	Да	Да			
ЛиБеллфорд лучше Беллмана-Форда	Да	Нет	Да			

Мы можем наблюдать то, что алгоритмы Дейкстры и Ли показали далеко не лучший результат. Ответ на вопрос “почему это так?” довольно прост - из-за большого количества блоков, которые нужно рассмотреть. На больших расстояниях данным алгоритмам требуется больше всего времени на построение пути (зато их путь действительно будет оптимальным)

Стоит отметить результативность гибридов: они смогли в итоге превзойти свои оригинальные, не модернизированные алгоритмы поиска

В состязании, когда задание было добраться до точки, удаленной от города на небольшое расстояние (вкладка OneItem), данные алгоритмы показали себя уже гораздо лучше:

Название алгоритма	1	2	3	Место	вклад	
Дейкстра	6513	7377	8169	6	8,61%	256197
Волновой (взвешенный граф)	11402	9574	9695	3	11,97%	
Волновой (не взвешенный граф)	4616	4990	4685	9	5,58%	
Поиск в ширину	4560	5105	4716	8	5,61%	
А-стар	9277	8605	8987	5	10,49%	
Диагональный	4424	5078	4789	9	5,58%	
Угловой	4579	4459	4712	11	5,37%	
Ликстра	8294	4200	7485	7	7,80%	
Листар	10144	8403	8878	4	10,70%	
Беллмана-Форда	13870	11811	10895	1	14,28%	
ЛиБеллФорд	12580	11292	12033	2	14,01%	

МИН находится в N строке	7	9	4
МАКС находится в N строке	11	11	12

Ликстра лучше Дейкстры	Да	Нет	Нет	
Листар лучше А*	Да	Нет	Нет	
ЛиБеллФорд лучше Беллмана-Форда	Нет	Нет	Да	

Гибриды, несмотря на то, что заточены на улучшение результата при дальних дистанциях, по разу но смогли обойти оригинальные алгоритмы поиска. В моментах, когда “нет” им не хватило небольшого количества очков.

Задание - доставка груза в некоторый город (вкладка TownDelivery), гарантированно находящийся в определенном радиусе от текущего города:

Название алгоритма	1	2	3	4	5	Место	вклад	
Дейкстра	17934	26446	17141	28826	26000	6	4,87%	1263078
Волновой (взвешенный граф)	22428	13802	15673	11160	6427	7	4,11%	
Волновой (не взвешенный граф)	8737	7205	17352	12136	10173	8	2,64%	
Поиск в ширину	8806	6592	17609	11866	10329	9	2,61%	
А-стар	22991	17191	47544	28658	21616	4	6,95%	
Диагональный	9330	6637	16676	12798	10575	10	2,58%	
Угловой	8685	6551	16716	11466	10521	11	2,53%	
Ликстра	23921	26073	24464	13297	13500	5	5,89%	
Листар	21440	16871	55051	30131	23412	3	7,39%	
Беллмана-Форда	41552	31159	72429	50812	39024	1	11,49%	
ЛиБеллФорд	39217	32317	71464	52619	39728	2	11,32%	

МИН находится в N строке	8	8	3	3	3
МАКС находится в N строке	11	12	11	12	12

Ликстра лучше Дейкстры	Да	Нет	Да	Нет	Нет
Листар лучше А*	Нет	Нет	Да	Да	Да
ЛиБеллФорд лучше Беллмана-Форда	Нет	Да	Нет	Да	Да

Вновь лидерство у алгоритма Беллмана-Форда и его гибрида. Доставка груза в города - задание, заточенное на то, что наилучшим путем

будет - пройти от одного города до другого по дороге, т.е. по самому быстрому маршруту. В этом тесте алгоритмы Дейкстры и A* показали вполне хорошие результаты. Особенно стоит отметить, что их гибриды смогли в итоге превзойти “родителей” за счет своей повышенной скорости построения маршрутов.

Гибриды (как и планировалось) наиболее хорошо проявили себя на задании, когда выдается случайная точка на карте, до которой необходимо дойти (вкладка RandomPoint), т.к. выданные точки могли находиться очень далеко и вдали от дороги (напомним, что проходимость дороги = 1):

Название алгоритма	1	2	3	Место	вклад	
Дейкстра	2983	1362	1935	10	2,15%	291686
Волновой (взвешенный граф)	0	0	0	11	0,00%	
Волновой (не взвешенный граф)	7754	6206	6811	8	7,12%	
Поиск в ширину	8570	6850	6547	6	7,53%	
А-стар	13402	10318	10068	4	11,58%	
Диагональный	8482	7053	6661	5	7,61%	
Угловой	8369	6403	6388	7	7,25%	
Ликстра	5252	3813	2798	9	4,07%	
Листар	13444	10542	10392	3	11,79%	
Беллмана-Форда	23890	20054	17619	1	21,11%	
ЛиБеллфорд	23222	16779	17719	2	19,79%	

МИН находится в N строке	3	3	3
МАКС находится в N строке	11	11	12

Ликстра лучше Дейкстры	Да	Да	Да
Листар лучше A*	Да	Да	Да
ЛиБеллфорд лучше Беллмана-Форда	Нет	Нет	Да

Особенно хорошо заметно, как волновой алгоритм не смог справиться с поставленной задачей (скорее всего за счет того, что выданная точка находилась слишком далеко, например на другом конце карты)

Самый большой тест:

Задание - дойти до случайной точки. Прошло 1.500.000 ЕВ (вкладка RandomPoint 1.5M ticks)

Название алгоритма	1	Место	вклад	
Дейкстра	13974	10	1,23%	1139789
Волновой (взвешенный граф)	2940	11	0,26%	
Волновой (не взвешенный граф)	108618	7	9,53%	
Поиск в ширину	112576	4	9,88%	
А-стар	105937	8	9,29%	
Диагональный	110581	5	9,70%	
Угловой	109663	6	9,62%	
Ликстра	20638	9	1,81%	
Листар	171921	2	15,08%	
Беллмана-Форда	154556	3	13,56%	
ЛиБеллФорд	228385	1	20,04%	

МИН находится в N строке	3
МАКС находится в N строке	12

Ликстра лучше Дейкстры	Да
Листар лучше А*	Да
ЛиБеллФорд лучше Беллмана-Форда	Да

Результаты поражают! Диагональный и угловой алгоритмы смогли обогнать Дейкстру и даже А*. И, если неудачу А* можно еще списать, что ему повезло с точками чуть меньше, то причина поражения Дейкстры - долгое построение пути.

Также стоит отметить, что на большом значении ЕВ гибриды показали себя в полной мере. Особенно хорошо видно результативность ЛиБеллФорда и Листара, которые набрали намного больше своих "родителей". Их вклад в общую копилку очков оказался на 6-7% выше.

762.000 ЕВ, задание - путешествие по городам (вкладка TownTravel 762.000 ticks)

Название алгоритма	1	Место	вклад	
Дейкстра	11085	9	2,27%	488953
Волновой (взвешенный граф)	2100	11	0,43%	
Волновой (не взвешенный граф)	39168	6	8,01%	
Поиск в ширину	39699	5	8,12%	
А-стар	50468	4	10,32%	
Диагональный	39140	7	8,00%	
Угловой	38326	8	7,84%	
Ликстра	10764	10	2,20%	
Листар	65307	3	13,36%	
Беллмана-Форда	89766	2	18,36%	
ЛиBellFord	103130	1	21,09%	
МИН находится в N строке		3		
МАКС находится в N строке		12		
Ликстра лучше Дейкстры	Нет			
Листар лучше А*	Да			
ЛиBellFord лучше Беллмана-Форда	Да			

Вновь заметно превосходство гибридов при больших значениях ЕВ. Однако Ликстра немного не добрала до Дейкстры, хотя в прошлом тесте гибрид имел превосходство. Почему? Причина в том, что до дальних точек да еще и таких, что они отдалены от дороги, гибрид построит путь быстрее.

Выводы

Выводы основывались на результатах тестов. Особое внимание хочется обратить на то, что построение пути и перемещение ботов по карте - вещи НЕ синхронизированные. Именно поэтому мы разбиваем алгоритмы на 2 группы: для взвешенного графа и не взвешенного.

Также стоит отметить, что при условии, что все боты будут ждать, пока другие алгоритмы построят пути, а потом уже одновременно выдвинутся - результаты будут совершенно другими. Как пример, алгоритм Дейкстры наберет наибольшее количество очков в данном случае.

Однако в нашей программе чем дольше строить путь - тем больше других алгоритмов уже дойдут до точки назначения и начнут строить следующие пути. Рассмотрим алгоритмы именно на данном случае.

Алгоритмы для взвешенного графа:

Алгоритм Дейкстры: хороший в плане быстрого пути (строит такой маршрут, что путь по нему займет наименьшее количество времени). Проблемы выявляются при больших расстояниях - алгоритму нужно немало времени, чтобы найти путь. В худшем случае данный алгоритм перебирает все клетки карты.

Алгоритм A*: строит путь быстрее Дейкстры. В некоторых случаях по оптимальности не уступает Дейкстре. Все зависит от функции, которая заложена в A*. Для поставленных задач данный алгоритм показывает себя лучше предшественника.

Волновой алгоритм (алгоритм Ли): для небольших расстояний подходит хорошо (в некоторых тестах даже лучше, чем Дейкстра и A*), однако для дальних дистанций - несомненно наихудший вариант из всех (в тестах можно заметить, что иногда данный алгоритм не успевал набрать и 1 очка за все время, в других случаях - набирал много меньше других)

Алгоритм Беллмана-Форда: в данном варианте его реализации оказался, пожалуй, лучшим решением для поставленных задач (по результатам тестов занимал 1-2 места, в одном случае - 1 и 3 места).

Гибриды, как и ожидалось, привели к улучшению результатов: их итоговый вклад был выше своих "родителей". В случаях, где гибрид не набирал больше очков составляющих его алгоритмов, ему не хватало небольших значений, т.е. разрыв был относительно мал.

Лучший: Беллман-Форд и его гибрид ЛиБеллФорд
Худший: Волновой алгоритм

Алгоритмы для не взвешенного графа:

Уголком: алгоритм, который выбирает путь по самой большой дуге - идет "уголком". Наибольшее удивление вызвало его 5 место при teste AllQuests, но подобный результат легко обосновывается - проигравшие ему алгоритмы слишком долго строят свои пути, что "уголком" пройти в итоге оказалось быстрее.

По диагонали - алгоритм, для которого нет преград, есть только цель. В случае, если бы клетки имели одну проходимость, мог бы оказаться самым быстрым из всех. Заслуженное место - сразу перед "уголком".

В ширину: алгоритм легко реализуемый, дающий средний результат.

Именно середины и придерживался в наших тестах

В глубину: самый неоптимальный вариант для построения маршрутов. В самых редких случаях выдает хорошие пути. В основном - оставляет желать лучшего. Худший из всех представленных алгоритмов

Волновой алгоритм (алгоритм Ли): аналогичен алгоритму в ширину. за счет этого и выдает примерно такие же результаты. Отличен лишь в построении пути обратно, отсюда и погрешность +- по очкам.

Лучший: в ширину за счет своей стабильности в выдаваемых результатах в виде средних показателей среди ВСЕХ алгоритмов

Худший: в глубину