



Илья Захаркин @QuantZero

23,0

карма

0,0

рейтинг

Профиль

1
Публикации2
Комментарии39
Избранное6
Подписчики

5 сентября в 18:58

Разработка → Диаграмма Вороного и её применения

из песочницы

Алгоритмы*, C++*

Доброго всем времени суток, уважаемые посетители сайта Хабрахабр. В данной статье я бы хотел рассказать вам о том, что такое диаграмма Вороного (изображена на картинке ниже), о различных алгоритмах её построения (за $O(n^4)$, $O(n^2 * \log(n))$ — пересечение полуплоскостей, $O(n * \log(n))$ — алгоритм Форчуна) и некоторых тонкостях реализации (на языке C++).



Также будет рассмотрено много интересных применений диаграммы и несколько любопытных фактов о ней. Будет интересно!

План статьи:

- [Необходимые понятия и определения](#)
- [Алгоритмы построения](#)
- [Применения](#)
- [Интересные факты](#)
- [Список литературы](#)

Стоит отметить, что в данной статье будут рассматриваться только алгоритмы построения диаграммы Вороного на плоскости. Попутно будут рассмотрены некоторые другие алгоритмы, необходимые для построения диаграммы — алгоритм определения точки пересечения двух отрезков, алгоритм О`Рурка пересечения двух выпуклых многоугольников, алгоритм построения «береговой линии».

Необходимые понятия и определения

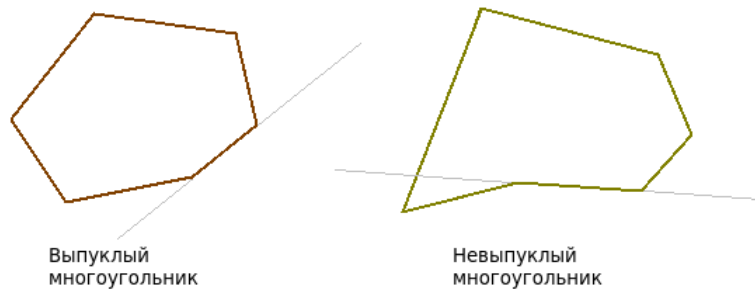
Сразу скажу, что **всё**, что будет дальше происходить, находится **на плоскости**.

Что ж, перед тем, как начать разбираться, что это такое — диаграмма Вороного, напомним некоторые понятия нужных нам геометрических объектов (предполагается, однако, что вы уже знакомы с определениями точки, прямой, луча, отрезка, многоугольника, вершины и ребра многоугольника, вектора и интуитивного понятия разбиения плоскости):

Простой многоугольник — это многоугольник без самопересечений. Мы будем рассматривать только простые многоугольники.

Невыпуклый многоугольник — это многоугольник, в котором найдутся такие две вершины, что через них проводится прямая, пересекающая данный многоугольник где-либо ещё, кроме ребра, соединяющего эти вершины (см. картинку),

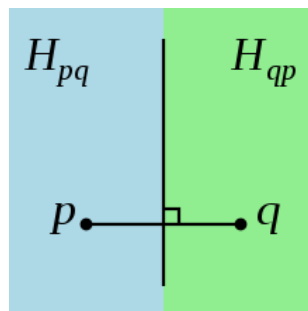
Выпуклый многоугольник — это многоугольник, у которого продолжения сторон не пересекают других его сторон (см. картинку). С другими вариантами определений можно ознакомиться на [википедии](#).



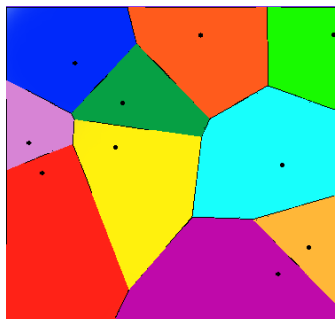
Именно из выпуклых многоугольников и будет состоять диаграмма. Почему именно из выпуклых? Потому что они являются ничем иным, как пересечением полуплоскостей (как мы увидим чуть позже), которые являются выпуклыми фигурами, а вот почему пересечение выпуклых фигур — выпуклая фигура, предложу вам узнать самим (доказательство есть, например, в книге [2]).

Раз уж я начал говорить про полуплоскости, то можно плавно перейти и к самой диаграмме — она состоит из так называемых **локусов** — областей, в которых присутствуют все точки, которые находятся ближе к данной точке, чем ко всем остальным. В диаграмме Вороного локусы являются выпуклыми многоугольниками.

Как строить **локус**? По определению он будет строиться так: пусть дано множество из n точек, для которого мы строим диаграмму. Возьмём конкретную точку p , для которой строим локус, и ещё одну точку из данного нам множества — q (не равную p). Проведём отрезок, соединяющий эти две точки, и проведём прямую, которая будет являться серединным перпендикуляром данного отрезка. Эта прямая делит плоскость на две полуплоскости — в одной лежит точка p , в другой лежит точка q . В данном случае локусами этих двух точек являются полученные полуплоскости. То есть для того, чтобы построить локус точки p , нужно получить пересечение всех таких полуплоскостей (то есть на месте q побывают все точки данного множества, кроме p).



Точку, для которой строится локус, называют **сайтом (site)**. На следующей картинке локусы помечены разными цветами.



Алгоритмы построения диаграммы как раз и есть не что иное, как алгоритмы построения этих самых *локусов* для всех точек из заданного набора. Локусы в данной задаче также называют *многоугольниками Вороного* или *ячейками Вороного*.

Наконец, сформулируем определение **диаграммы Вороного n точек на плоскости** (n — натуральное) — это **разбиение плоскости**, состоящее из n **локусов** (для каждой точки по локусу). Опять же, другой вариант определения можно найти на [википедии](#).

Кстати, вот [сайт с интерактивным цветным визуализатором](#) диаграммы Вороного, можно самому нажимать на область и **заливать** видеть, как строится диаграмма.

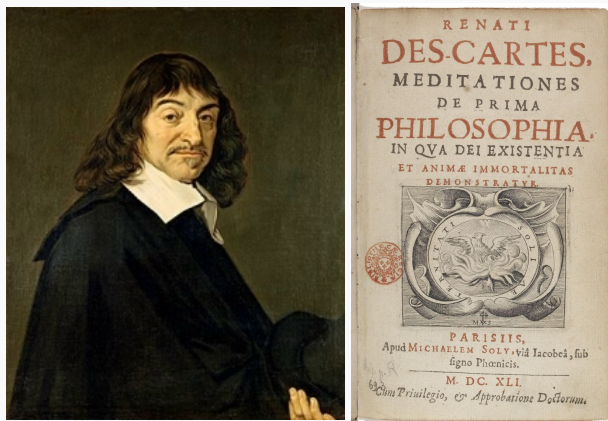
Если интересно, как появилась диаграмма и почему она носит фамилию Вороного, то вам стоит взглянуть под спойлер ниже.

Исторические факты

Немного истории

(материал взят с [этого](#) сайта)

Вообще, первое использование этой диаграммы встречается в труде Рене Декарта (1596-1650) «Начала философии» (1644). Декарт предложил деление Вселенной на зоны гравитационного влияния звезд.



Только спустя два века, известный немецкий математик Иоганн Петер Густав Лежён-Дирихле (1805 — 1859) ввел диаграммы для двух- и трехмерного случаев. Поэтому их иногда называют диаграммами Дирихле.



Ну а уже в 1908 году русский математик Георгий Феодосьевич Вороной (16(28) апреля 1868 — 7(20) ноября 1908) описал эту диаграмму для пространств БОльших размерностей, с тех пор диаграмма носит его фамилию. Вот его краткая биография (взято из [википедии](#)):

Георгий Феодосьевич Вороной родился в деревне Журавка Полтавской губернии (ныне Черниговская область). С 1889 года обучался в Санкт-Петербургском университете у *Андрея Маркова*. В 1894 году защитил магистерскую диссертацию «О целых числах, зависящих от корня уравнения третьей степени». В том же году был избран профессором Варшавского университета, где изучал цепные дроби. У Вороного обучался *Вацлав Серпинский*. В 1897 году Вороной защитил докторскую диссертацию «Об одном обобщении алгоритма непрерывных дробей», удостоенную премии имени Буняковского.



Алгоритмы построения

Научимся строить диаграмму Вороного. Мы рассмотрим 4 алгоритма, 2 из которых подробно (1 с реализацией, полной реализации алгоритма Форчуна будет посвящена отдельная статья), ещё 2 кратко (и без реализации):

1. Алгоритм построения диаграммы Вороного «в лоб». Сложность: $O(n^4)$;
2. Алгоритм построения диаграммы Вороного путём пересечения полуплоскостей. Сложность: $O(n^2 * \log(n))$;
3. Алгоритм Форчуна построения диаграммы Вороного на плоскости. Сложность: $O(n * \log(n))$;
4. Рекурсивный алгоритм построения диаграммы Вороного. Сложность: $O(n * \log(n))$.

После описания некоторых алгоритмов будет приведена его реализация на языке C++. Для реализации использовалась написанная нами библиотека SplashGeom © — [ссылка на github](#), в которой есть всё необходимое для реализации многих алгоритмов вычислительной геометрии на плоскости и некоторых в пространстве. Прошу не судить строго данную библиотеку, она ещё находится в стадии активной разработки и улучшения, однако все замечания будут услышаны.

Если же вам интересны другие реализации, то вот ещё некоторые:

- [boost.polygon](#)
- [Сайт Стива Форчуна](#)
- [aewllin/openvoronoi](#)
- [Реализация на java](#)
- [Реализация на JavaScript \(там ещё и визуализатор есть\)](#)

Теперь перейдём к непосредственному рассмотрению алгоритмов:

Алгоритм построения диаграммы Вороного «в лоб» за $O(n^4)$

Здесь идея в том, чтобы пересекать не полуплоскости, а именно серединные перпендикуляры отрезков (потому что это проще, согласитесь), соединяющих данную точку со всеми другими точками. То есть мы, следуя определению ячейки Вороного, будем строить locus для точки **p** так:

1. Получаем $n-1$ прямую (серединные перпендикуляры), так как мы провели серединные перпендикуляры всех отрезков, соединяющих данную точку **p** с остальными;
2. Пересекаем попарно все прямые, получаем $O(n^2)$ точек пересечения (потому что каждая прямая может пересечь все другие, в «худшем случае»);
3. Проверяем все эти $O(n^2)$ точек на принадлежность каждой из $n-1$ полуплоскостей, то есть получаем уже асимптотику $O(n^3)$. Соответственно те точки, которые принадлежат всем полуплоскостям, и будут вершинами ячейки Вороного точки **p**;
4. Прodelываем первые три шага для всех n точек, получаем итоговую асимптотику $O(n^4)$.

С алгоритмом также можно ознакомиться на [e-maxx.ru](#).

При желании Вы самостоятельно можете реализовать этот алгоритм с помощью SplashGeom ©. В данной статье его реализация не приводится, потому как данный алгоритм на практике сильно уступает хотя бы следующему...

Алгоритм построения диаграммы Вороного путём пересечения полуплоскостей за $O(n^2 * \log(n))$

Этот алгоритм уже можно использовать на практике, так как он не имеет столь большой вычислительной сложности. Для него нам потребуется: уметь пересекать отрезки и прямые, уметь пересекать выпуклые многоугольники, уметь пересекать полуплоскости, уметь объединять полученные locus в диаграмму.

Алгоритм

1. Получаем $n-1$ прямую для текущего сайта (как в предыдущем алгоритме — серединные перпендикуляры). Это будут «образующие» полуплоскостей;
2. Теперь мы имеем $n-1$ полуплоскость. Каждая из этих полуплоскостей задаётся какой-либо прямой из пред. пункта и ориентацией, то есть с какой стороны от прямой она расположена. Ориентацию можно определить по текущему сайту, для которого строим locus — он лежит в искомой полуплоскости, а значит и его locus должен лежать в ней;
3. Пересекаем все полуплоскости — мы сможем делать это за $O(n * \log(n))$ — получаем locus для текущего сайта;
4. Прodelываем первые три шага для всех n точек, получаем итоговую асимптотику $n * O(n * \log(n)) = O(n^2 * \log(n))$.

Реализация

Основная загвоздка здесь, я считаю, реализовать нормальное пересечение выпуклых многоугольников, потому как там есть неприятные вырожденные случаи (совпадение вершин и/или сторон многоугольников; в случае пересечения многоугольника с самим собой надо неплохо подумать, чтобы адаптировать алгоритм О`Рурка для корректной работы в этом случае).

Сразу скажу, что мы ограничиваем всю область действий **ограничивающим прямоугольником** — полуплоскости будут

отсекать от него части, которые мы и пересечём друг с другом. Это решает проблему бесконечных ячеек в диаграмме.

Пересечение прямых и отрезков

Причём нам нужны именно точки пересечения, а не просто определение его наличия. В SplashGeom © это есть — вот, к примеру, реализация пересечения прямых и отрезков:

▼ [Посмотреть код](#)

```
// пересечение прямых
// kInfPoint - прямые параллельны, kNegInfPoint - прямые совпадают
Point2D Line2D::GetIntersection(const Line2D& second_line) const
{
    double cross_prod_norms = Vector2D(this->A, this->B).OrientedCCW(Vector2D(second_line.A, second_line.B));
    Point2D intersect_point;
    if (fabs(cross_prod_norms) <= EPS) /* A1 / A2 == B1 / B2 */ {
        if (fabs(this->B * second_line.C - second_line.B * this->C) <= EPS) /* .. == C1 / C2 */ {
            intersect_point = kNegInfPoint2D;
        } else {
            intersect_point = kInfPoint2D;
        }
    } else {
        double res_x = (second_line.C * this->B - this->C * second_line.B) / cross_prod_norms;
        double res_y = (second_line.A * this->C - this->A * second_line.C) / cross_prod_norms;
        intersect_point = Point2D(res_x, res_y);
    }
    return intersect_point;
}

// пересечение отрезков
Point2D Segment2D::GetIntersection(const Segment2D& second_seg) const
{
    Line2D first_line(*this);
    Line2D second_line(second_seg);
    Point2D intersect_point = first_line.GetIntersection(second_line);
    if (intersect_point == kNegInfPoint2D) {
        if (this->Contains(second_seg.b)) {
            intersect_point = second_seg.b;
        } else if (this->Contains(second_seg.a)) {
            intersect_point = second_seg.a;
        } else if (second_seg.Contains(this->b)) {
            intersect_point = this->b;
        } else if (second_seg.Contains(this->a)) {
            intersect_point = this->a;
        } else {
            intersect_point = kInfPoint2D;
        }
    } else if (!(this->Contains(intersect_point) && second_seg.Contains(intersect_point))) {
        intersect_point = kInfPoint2D;
    }
    return intersect_point;
}
```

Пересечение выпуклых многоугольников

Теперь реализуем пересечение многоугольников. Можно, конечно, делать это за $O(nm)$, где n и m — количество вершин первого и второго многоугольника соответственно — пересекать каждую сторону первого многоугольника с каждой стороной из второго, записывая точки пересечения, а также проверять точки на принадлежность другому многоугольнику, но мы, дабы достичь лучшей скорости, будем пересекать по алгоритму О`Рурка (оригинальное описание алгоритма — [3]).

Также с одним из вариантов его описания можно ознакомиться на algotlist.ru. У нас же реализация основана на описании алгоритма в книге [1] (стр. 334), с некоторыми дополняющими идеями. Сразу отмечу, что данная реализация **не учитывает случаи**, когда у многоугольников **есть общие стороны** (как отмечено в книге [1], этот случай требует отдельного рассмотрения), однако случаи с общими вершинами работают корректно.

Под спойлером ниже можно увидеть общее описание алгоритма.

▼ Хочу знать алгоритм О`Рурка!

Алгоритм (за дополнительными пояснениями обратитесь к выше указанным источникам):

- Пока не пройдёт максимальное количество итераций (доказывается, что оно не больше $2 \cdot (n + m)$), выполняем следующие инструкции:
 - Берём текущие рёбра первого и второго многоугольников;
 - Если они пересекаются — тогда рассматриваем точку из пересечения: может быть так, что мы только что её добавили в многоугольник пересечения, тогда просто игнорируем добавление, иначе — если она не является начальной точкой пересечения (то есть мы уже сделали круг), то добавляем в пересечение, иначе заканчиваем алгоритм — мы встретили точку пересечения, которая была в начале;
 - Далее (независимо от того, пересекаются текущие рёбра или нет) вызываем функцию **Движение**, которая отвечает за передвижение окна первого (или второго) многоугольника на одно ребро вперёд, а также за возможное добавление вершины в многоугольник пересечения. Основные действия происходят именно в **Движении**.
- Если не было точек пересечения, определить, лежит ли один многоугольник внутри другого (проверка принадлежности точки выпуклому многоугольнику за $O(\log(\text{количество вершин многоугольника}))$ — [1], стр. 59-60). Если лежит, то вернуть его, иначе вернуть пустое пересечение.

Функцию **Движения** можно реализовывать по-разному, у нас она возвращает номер (метку) многоугольника, ребро которого мы двигаем. Концептуально она внутри себя делает три вещи:

- определяет **номер случая** — текущее взаимное расположение ребра первого многоугольника и ребра второго многоугольника. Это **основная идея алгоритма** — просмотр случаев положения рёбер относительно друг друга. Все эти четыре положения хорошо описаны в [1], в нашей реализации положение определяется с помощью косо́го произведения векторов на плоскости;
- определяет, ребро какого многоугольника сейчас «внутри», то есть оно лежит «слева» от другого ребра, у нас это тоже проверяется косым произведением (лежит «слева», если конечная точка лежит «слева»);
- решает, записывать ли текущий конец одного из рёбер в многоугольник пересечения. Если это соответствует нужному случаю и ребро внутри, то нужно добавить, иначе нет.

Ну а в SplashGeom © это выглядит так:

▼ Код пересечения многоугольников

```
// для получения многоугольников, которые полуплоскости отсекают от ограничивающего прямоугольника
Convex2D Rectangle::GetIntersectionalConvex2D(const Point2D& cur_point, const Line2D& halfplane) const
{
    vector<Point2D> convex_points;
    Segment2D cur_side;
    Point2D intersection_point;
    for (int i = 0, sz = vertices_.size(); i < sz; ++i) {
        int j = (i + 1) % sz;
        cur_side = Segment2D(vertices_[i], vertices_[j]);
        intersection_point = halfplane.GetIntersection(cur_side);
        if (intersection_point != kInfPoint2D)
            convex_points.push_back(intersection_point);
        if (halfplane.Sign(cur_point) == halfplane.Sign(vertices_[i]))
            convex_points.push_back(vertices_[i]);
    }
    Convex2D result_polygon(MakeConvexHullJarvis(convex_points));
    return result_polygon;
}

// определяет номер случая взаимного расположения рёбер
NumOfCase EdgesCaseNum(const Segment2D& first_edge, const Segment2D& second_edge)
{
    bool first_looks_at_second = first_edge.LooksAt(second_edge);
    bool second_looks_at_first = second_edge.LooksAt(first_edge);
    if (first_looks_at_second && second_looks_at_first) {
        return NumOfCase::kBothLooks;
    } else if (first_looks_at_second) {
        return NumOfCase::kFirstLooksAtSecond;
    } else if (second_looks_at_first) {
        return NumOfCase::kSecondLooksAtFirst;
    } else {
        return NumOfCase::kBothNotLooks;
    }
}
```

```

    }
}

// определяет, какое ребро сейчас "внутри"
WhichEdge WhichEdgeIsInside(const Segment2D& first_edge, const Segment2D& second_edge)
{
    double first_second_side = Vector2D(second_edge).OrientedCCW(Vector2D(second_edge.a, first_edge.b));
    double second_first_side = Vector2D(first_edge).OrientedCCW(Vector2D(first_edge.a, second_edge.b));
    if (first_second_side < 0) {
        return WhichEdge::kSecondEdge;
    } else if (second_first_side < 0) {
        return WhichEdge::kFirstEdge;
    } else {
        return WhichEdge::Unknown;
    }
}

// функция Движения
WhichEdge MoveOneOfEdges(const Segment2D& first_edge, const Segment2D& second_edge, Convex2D& result_polygon)
{
    WhichEdge now_inside = WhichEdgeIsInside(first_edge, second_edge);
    NumOfCase case_num = EdgesCaseNum(first_edge, second_edge);
    WhichEdge which_edge_is_moving;
    switch (case_num) {
        case NumOfCase::kBothLooks: {
            if (now_inside == WhichEdge::kFirstEdge) {
                which_edge_is_moving = WhichEdge::kSecondEdge;
            } else {
                which_edge_is_moving = WhichEdge::kFirstEdge;
            }
            break;
        }
        case NumOfCase::kFirstLooksAtSecond: {
            which_edge_is_moving = WhichEdge::kFirstEdge;
            break;
        }
        case NumOfCase::kSecondLooksAtFirst: {
            which_edge_is_moving = WhichEdge::kSecondEdge;
            break;
        }
        case NumOfCase::kBothNotLooks: {
            if (now_inside == WhichEdge::kFirstEdge) {
                which_edge_is_moving = WhichEdge::kSecondEdge;
            } else {
                which_edge_is_moving = WhichEdge::kFirstEdge;
            }
            break;
        }
    }

    if (result_polygon.Size() != 0 && (case_num == NumOfCase::kFirstLooksAtSecond || case_num == NumOfCase::kSecondLooksAtFirst)) {
        Point2D vertex_to_add;
        if (now_inside == WhichEdge::kFirstEdge) {
            vertex_to_add = first_edge.b;
        } else if (now_inside == WhichEdge::kSecondEdge) {
            vertex_to_add = second_edge.b;
        } else {
            if (case_num == NumOfCase::kFirstLooksAtSecond)
                vertex_to_add = first_edge.b; // ?!
            else
                vertex_to_add = second_edge.b;
        }
        if (vertex_to_add != result_polygon.GetCurVertex())
            result_polygon.AddVertex(vertex_to_add);
    }
    return which_edge_is_moving;
}

// пересечение выпуклых многоугольников (ссылка не константная только потому, что во втором многоугольнике двигается окно)

```

```

Convex2D Convex2D::GetIntersectionalConvex(Convex2D& second_polygon)
{
    Convex2D result_polygon;
    size_t max_iter = 2 * (this->Size() + second_polygon.Size());
    Segment2D cur_fp_edge; // current first polygon edge
    Segment2D cur_sp_edge; // current second polygon edge
    Point2D intersection_point;
    bool no_intersection = true;
    WhichEdge moving_edge = WhichEdge::Unknown;
    for (size_t i = 0; i < max_iter; ++i) {
        cur_fp_edge = this->GetCurEdge();
        cur_sp_edge = second_polygon.GetCurEdge();
        intersection_point = cur_fp_edge.GetIntersection(cur_sp_edge);
        if (intersection_point != kInfPoint2D) {
            if (result_polygon.Size() == 0) {
                no_intersection = false;
                result_polygon.AddVertex(intersection_point);
            } else if (intersection_point != result_polygon.GetCurVertex()) {
                if (intersection_point == result_polygon.vertices[0]) {
                    break; // we already found the intersection polygon
                } else {
                    result_polygon.AddVertex(intersection_point);
                }
            }
        }
        moving_edge = MoveOneOfEdges(cur_fp_edge, cur_sp_edge, result_polygon);
        if (moving_edge == WhichEdge::kFirstEdge) {
            this->MoveCurVertex();
        } else {
            second_polygon.MoveCurVertex();
        }
    }
    if (no_intersection == true) {
        if (second_polygon.Contains(this->GetCurVertex())) {
            result_polygon = *this;
        } else if (this->Contains(second_polygon.GetCurVertex())) {
            result_polygon = second_polygon;
        }
    }
    return result_polygon;
}

```

Надеюсь, данное описание и код будут вам полезны.

Пересечение полуплоскостей

Итак, у нас есть всё необходимое для построения пересечения полуплоскостей. Что ж, сделаем это, да сделаем не просто, а по-умному — в силу ассоциативности операции пересечения полуплоскостей, мы можем пересекать их в любом порядке, а значит пересечём две плоскости, потом пересечём ещё две, а потом пересечём их пересечения, это ведь уже быстрее, чем пересекать все по-отдельности.

Так что тут целесообразно использовать *рекурсию* (~~сразу написание испортилось~~) — её работа здесь вполне понятна, сами посмотрите:

▼ [Посмотреть код](#)

```

Convex2D GetHalfPlanesIntersection(const Point2D& cur_point, const vector<Line2D>& halfplanes, const Rectangle& border_box)
{
    if (halfplanes.size() == 1) {
        Convex2D cur_convex(border_box.GetIntersectionalConvex2D(cur_point, halfplanes[0]));
        return cur_convex;
    } else {
        int middle = halfplanes.size() >> 1;
        vector<Line2D> first_half(halfplanes.begin(), halfplanes.begin() + middle);

```

```

vector<Line2D> second_half(halfplanes.begin() + middle, halfplanes.end());
Convex2D first_convex(GetHalfPlanesIntersection(cur_point, first_half, border_box);
Convex2D second_convex(GetHalfPlanesIntersection(cur_point, second_half, border_box);
return first_convex.GetIntersectionalConvex(second_convex);
}
}

```

Добавление локуса в диаграмму

Теперь мы имеем многоугольник Вороного данной точки, пора добавить его в диаграмму. Проблем с этим здесь особо не возникает, потому как мы получаем области в произвольном порядке, и между собой они никак не связаны (в отличие от реализации в алгоритме Форчуна, где можно перейти на соседний локус по указателю на ребро):

▼ [Посмотреть код](#)

```

// строим локус для текущего сайта
Voronoi2DLocus VoronoiDiagram2D::MakeVoronoi2DLocus(const Point2D& site, const vector<Point2D>& points, const Rectangle& border_box)
{
    Voronoi2DLocus cur_locus;
    vector<Line2D> halfplanes;
    for (auto cur_point : points) {
        if (cur_point != site) {
            Segment2D cur_seg(site, cur_point);
            Line2D cur_halfplane(cur_seg.GetCenter(), cur_seg.NormalVec());
            halfplanes.push_back(cur_halfplane);
        }
    }
    *cur_locus.region_ = GetHalfPlanesIntersection(site, halfplanes, border_box);
    cur_locus.site_ = site;
    return cur_locus;
}

// строим диаграмму Вороного
VoronoiDiagram2D VoronoiDiagram2D::MakeVoronoiDiagram2DHalfPlanes(const vector<Point2D>& points, const Rectangle& border_box)
{
    Voronoi2DLocus cur_locus;
    for (auto cur_point : points) {
        cur_locus = MakeVoronoi2DLocus(cur_point, points, border_box);
        this->diagram_.push_back(cur_locus);
    }
    return *this;
}

```

Так, мы научились строить диаграмму Вороного за $O(n^2 * \log(n))$, она имеет вид вектора (списка) локусов. Недостаток данного решения — информации о соседях не получить (возможно, этот недостаток можно ликвидировать улучшенной реализацией).

Куда больше информации можно получить из РСДС (DCEL). Эта структура будет использована в алгоритме Форчуна.

Далее будет описан алгоритм Форчуна с использованием заметающей прямой и «береговой линии» (~~готовьте купальники и плажки — идем на пляж~~). По моему мнению, это самый приемлемый вариант, если вы хотите реализовать построение диаграммы Вороного и строить её со «скоростью» ~~света~~ $O(n * \log(n))$.

Алгоритм Форчуна построения диаграммы Вороного за $O(n * \log(n))$

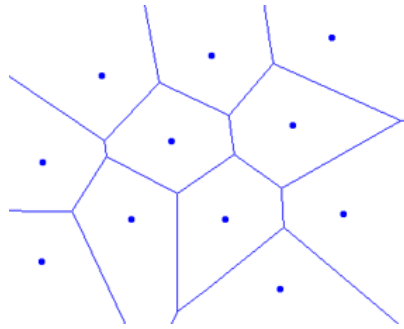
В 1987 году Стив Форчун (Steve Fortune) предложил алгоритм построения диаграммы за $O(n * \log(n))$. Конечно, он является не единственным алгоритмом построения с такой асимптотикой, но он достаточно понятен и не очень сложен в реализации (да и, к тому же, очень красив и математичен!), поэтому я выбрал именно его.

Материалы по алгоритму Форчуна можно найти [тут](#), [здесь](#), [вот здесь](#) и [ещё вот здесь](#).

Кстати, рассмотрению данного алгоритма уже была посвящена [статья на Хабрахабре](#).

Итак, основная идея алгоритма — это так называемая **заметаящая прямая (ЗП) (sweep line)**. Она применяется во многих алгоритмах вычислительной геометрии, потому что позволяет удобно моделировать движение прямой по некоторому множеству объектов (например, в алгоритме пересечения n отрезков тоже используется sweep line).

Перед тем, как начать говорить про то, как и что мы будем делать, давайте посмотрим, как движется sweep line (взято [отсюда](#)):



Красиво, не правда ли? В реализации всё примерно так же, только ЗП обычно движется сверху вниз, а не слева направо, и на самом деле всё не так плавно, а происходит от события к событию (см. ниже), то есть *дискретно*.

Суть алгоритма

Есть n сайтов (точек на плоскости). Есть заметаящая прямая, которая двигается (например) «сверху вниз», то есть от сайта с наибольшей ординатой к сайту с меньшей (от события к событию, если быть точным). Сразу стоит отметить, что влияние на построение диаграммы оказывают *только* те сайты, которые находятся *выше или на* заметающей прямой.

Когда ЗП попадает на очередной сайт (происходит **событие точки (point event)**), создаётся новая парабола (arch), **фокусом** которой является данный сайт, а **директрисой** — заметаящая прямая ([про параболу на википедии](#)). Эта парабола делит плоскость на две части — «внутренняя» область параболы соответствует точкам, которые сейчас ближе к сайту, а «внешняя» область — точкам, которые ближе к sweep line, ну а точки, лежащие на параболе — равноудалены от сайта и ЗП. Парабола будет меняться в зависимости от положения ЗП к сайту — чем дальше ЗП уходит от сайта вниз, тем больше расширяется парабола, однако в самом начале она вообще является отрезком («направленным» вверх).

Далее парабола расширяется, у неё появляются две **контрольные точки (break points)** — точки её пересечения с остальными параболой («береговой линией»). В «береговой линии» мы храним дуги парабол от одной точки пересечения их друг с другом до другой, так и получается beach line. По сути, в этом алгоритме мы моделируем движение этой «береговой линии». Потому как эти самые break point`ы движутся аккуратно по рёбрам ячеек Вороного (ведь получается, что контрольные точки равноудалены от обоих сайтов, которым соответствуют эти параболы, да ещё и от ЗП).

И как раз-таки в тот момент, когда две контрольные точки — по одной из разных парабол — «встречаются», то есть как бы превращаются в одну, эта точка и становится вершиной ячейки Вороного (происходит **событие круга (circle event)**), причём в это время та дуга, которая находилась между этими двумя точками — «схлопывается» и удаляется из «береговой линии». Далее мы просто соединяем эту точку с предыдущей соответствующей ей и получаем ребро ячейки Вороного.

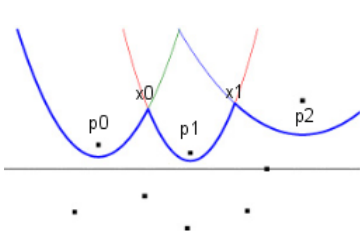
Алгоритм

Итак, при движении sweep line вниз мы встречаемся с двумя *типами событий*:

Событие точки (point event)

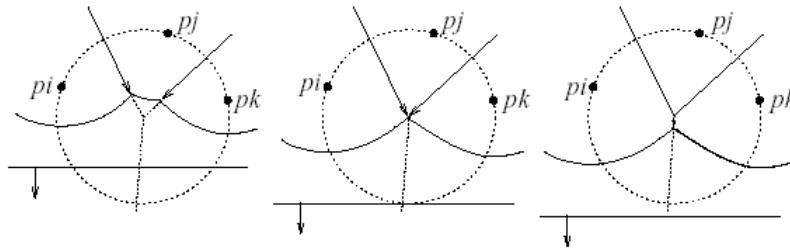
Событие точки — это попадание ЗП на один из сайтов, поэтому мы создаём новую параболу, соответствующую данному сайту, а также добавляются две **контрольные точки (break points)** (на самом деле сначала — одна, а при расширении арки уже две) — точки пересечения этой параболы с береговой линией (то есть с фронтом уже существующих парабол). Стоит отметить, что в данном алгоритме парабола (а точнее её часть, принадлежащая «береговой линии» — **арка**) «вставляется в береговую линию» **только** в случае события точки, то есть новая арка может появиться только при обработке **события точки**.

Кстати, на следующих картинках видно, почему объединения таких «кусков парабол» называют «береговой линией».



Событие круга (circle event)

Событие круга — это возникновение новой вершины ячейки Вороного вместе с удалением одной арки, потому что возникновение новой вершины диаграммы здесь означает, что было три арки, левая, средняя и правая, средняя «схлопывается» вследствие сближения левой и правой точек арок и получается новая вершина диаграммы Вороного. Стоит отметить, что в данном алгоритме парабола (арка) удаляется из «береговой линии» **только** в случае события круга, то есть арка может удалиться только при обработке **события круга**.



Есть теорема, в которой говорится, что вершина диаграммы Вороного всегда лежит на пересечении ровно трёх рёбер диаграммы, и есть следствие из этой теоремы, которое гласит, что вершина диаграммы является центром окружности, проходящей через три сайта и расстояние от этой точки до заметающей прямой тоже равно радиусу этой окружности (это свойство точек, лежащих на «береговой линии»). Это — **ключевой момент**, потому что именно когда **самая нижняя точка окружности**, проходящей через три сайта, лежит **ниже или на** заметающей прямой, мы пушим в очередь событий событие круга с этой самой нижней точкой, потому что когда прямая на неё попадёт, мы получим вершину диаграммы Вороного.

Важно, что с любым событием (точки или круга) связана одна конкретная арка, и наоборот. Это пригодится при обработке событий. Также надо не забыть, что нужно вовремя добавлять рёбра в РСДС (DCEL) (пункт 1 в структурах, см. ниже), так что надо понимать связь арок с рёбрами.

Таким образом, движение прямой дискретно — прямая в любой момент времени **либо на сайте, либо в нижней точке окружности**, проходящей через три сайта, центр которой — новая вершина диаграммы Вороного. Прекрасно.

Общий алгоритм:

1. Создаём очередь (с приоритетом) событий, изначально инициализируя событиями точки — данным множеством сайтов (ведь каждому сайту соответствует событие точки);
2. Пока очередь не пуста:
 - а). Берём из неё событие;
 - б). Если это — *событие точки*, то *обрабатываем событие точки*;
 - в). Если это — *событие круга*, то *обрабатываем событие круга*;
3. Закончить все оставшиеся рёбра (поработать с border_box).

Реализация

Реализация алгоритма Форчуна *будет рассмотрена подробно в отдельной статье*, однако здесь приводятся некоторые наработки, которые могут помочь в его понимании.

Необходимые структуры

Для реализации данного алгоритма нам понадобятся несколько структур (классов), а именно:

- РСДС (DCEL) — список для хранения уже найденный рёбер диаграммы Вороного;
- Приоритетная очередь с событиями;
- Двоичное дерево (у нас это BeachSearchTree) — для хранения «береговой линии» — текущего положения парабол и точек. Стоит отметить, что это дерево является сбалансированным — у узла либо ровно два сына, либо ноль (является листом).

Подробнее об этой структуре можно прочитать, например, в [этой статье на Хабре](#) (у нас она представлена немного по-другому).

Имея такие структуры данных, можем написать реализацию общего алгоритма:

▼ Посмотреть код

```
VoronoiDiagram2D VoronoiDiagram2D::MakeVoronoiDiagram2DFortune(const vector<Point2D>& points, const Rectangle& border_box)
{
    priority_queue<Event> events_queue(points.begin(), points.end());
    shared_ptr<Event> cur_event;
    BeachSearchTree beach_line;
    DCEL edges;
    while (!events_queue.empty()) {
        cur_event = make_shared<Event>(events_queue.top());
        shared_ptr<const PointEvent> is_point_event(dynamic_cast<const PointEvent *>(cur_event.get()));
        if (is_point_event) {
            events_queue.pop();
            beach_line.HandlePointEvent(*is_point_event, border_box, events_queue, edges);
        } else {
            shared_ptr<const CircleEvent> is_circle_event(dynamic_cast<const CircleEvent *>(cur_event.get()));
            events_queue.pop();
            beach_line.HandleCircleEvent(*is_circle_event, border_box, events_queue, edges);
        }
    }
    edges.Finish(border_box);
    this->dcel_ = edges;
    return *this;
}
```

Вся логика и сложность в HandlePointEvent() и HandleCircleEvent(), им и будет посвящена отдельная статья, далее же приведу некоторые вспомогательные функции, которые потом помогут в реализации.

Вспомогательные функции

Пересечение парабол (арок)

Нам нужно уметь получать пересечение двух парабол (арок) в зависимости от положения ЗП. Уравнение параболы с фокусом в точке x' и y' и директрисой, положение которой по оси y равно l , задаётся следующим уравнением (его можно вывести):

$$y = \frac{1}{2(y' - l)}((x - x')^2 + y'^2 - l^2)$$

Кстати, дробь перед скобкой — это *фокальный параметр* данной параболы. Отсюда мы можем «вытащить» соответствующие коэффициенты уравнения параболы и решить систему из двух нелинейных уравнений простым способом — вычтем одно из другого, а потом подставим найденные корни в первое, получим две точки. Нас интересует та точка, которая ниже (то есть с меньшей ординатой), потому что высокая точка лежит за «береговой линией». Описанные действия отражаются в следующем коде:

▼ Код пересечения парабол

```
pair<Point2D, Point2D> Arch::GetIntersection(const Arch& second_arch, double line_pos) const
{
    pair<Point2D, Point2D> intersect_points;
    double p1 = 2 * (line_pos - this->focus_->y);
    double p2 = 2 * (line_pos - second_arch.focus_->y); // is not 0.0, because line moved down
    if (fabs(p1) <= EPS) {
        intersect_points.first = this->GetIntersection(Ray2D(this->focus_, Point2D(this->focus_->x, this->focus_->y + 1)));
        intersect_points.second = intersect_points.first;
    } else {
        // solving the equation
        double a1 = 1 / p1;
        double a2 = 1 / p2;
```

```

double a = a2 - a1;
double b1 = -this->focus_->x / p1;
double b2 = -second_arch.focus_->x / p2;
double b = b2 - b1;
double c1 = pow(this->focus_->x, 2) + pow(this->focus_->y, 2) - pow(line_pos, 2) / p1;
double c2 = pow(second_arch.focus_->x, 2) + pow(second_arch.focus_->y, 2) - pow(line_pos, 2) / p1;
double c = c2 - c1;
double D = pow(b, 2) - 4 * a * c;
if (D < 0) {
    intersect_points = make_pair(kInfPoint2D, kInfPoint2D);
} else if (fabs(D) <= EPS) {
    double x = -b / (2 * a);
    double y = a1 * pow(x, 2) + b1 * x + c1;
    intersect_points = make_pair(Point2D(x, y), Point2D(x, y));
} else {
    double x1 = (-b - sqrt(D)) / (2 * a);
    double x2 = (-b + sqrt(D)) / (2 * a);
    double y1 = a1 * pow(x1, 2) + b1 * x1 + c1;
    double y2 = a1 * pow(x2, 2) + b1 * x2 + c1;
    intersect_points = make_pair(Point2D(x1, y1), Point2D(x2, y2));
}
return intersect_points;
}

```

Построение окружности по трём точкам

При обработке события круга нам понадобится определять центр и самую нижнюю точку окружности, построенной по фокусам трёх арок (сайтам). Есть некоторые аналитические алгоритмы построения окружности по трём точкам (под построением мы понимаем получение её центра и радиуса), у нас в программе это сделано так ([спасибо алголисту](#)) — соединяем отрезками первые две точки и вторые две точки. Центр лежит на пересечении серединных перпендикуляров, радиус — расстояние от центра до любой из трёх точек. Быстро и красиво:

▼ Посмотреть код построения окружности по трём точкам

```

Circle::Circle(const Point2D& p1, const Point2D& p2, const Point2D& p3)
{
    Segment2D first_segment(p1, p2);
    Segment2D second_segment(p2, p3);
    Line2D first_perpendicular(first_segment.GetCenter(), first_segment.NormalVec());
    Line2D second_perpendicular(second_segment.GetCenter(), second_segment.NormalVec());
    center_ = first_perpendicular.GetIntersection(second_perpendicular);
    little_haxis_ = big_haxis_ = center_.l2_distance(p1);
}

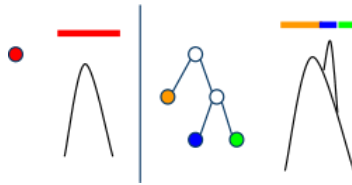
```

Обработка события точки

Событие точки — это когда мы вытащили из очереди PointEvent. Что в нём есть? В нём есть только сайт, с которым ассоциируется это событие.

Что мы делаем при его обработке? Мы добавляем новую арку в «береговую линию», настраивая в дереве все связи «как надо», и проверяем, не появилось ли событие круга в одном из трёх возможных случаев — нам нужно проверить все случаи, в которых может участвовать новый сайт.

Что мы делаем при добавлении арки? Мы ищем для неё место в нашем двоичном дереве «береговой линии» (по координате x), затем вставляем её.



Что мы делаем при вставке? Мы нашли указатель на арку, с которой новая имеет пересечение в двух точках (случай с попаданием точки пересечения ровно на одну из контрольных точек рассматривается отдельно — там пересечение будет с двумя параболой — слева и справа).

То есть мы берём эту арку, которая «разбивается», и вместо неё вставляем аж 5 узлов (был 1, стало 5, да) — arch1, bp1, arch2, bp2, arch3. Arch1 — это левый кусок арки, которую пересекает новая, то есть это кусок слева от левого break point `a, bp1 — левый break point (левое пересечение новой арки), arch2 — это новая арка собственной персоной, bp2 — правый break point (правое пересечение новой арки), arch3 — это правый кусок арки, которую пересекает новая.

Стоит отметить, что событие точки даёт начало новому ребру (или рёбрам, есть случаи) диаграммы Вороного.

Один из возможных вариантов добавления новой арки в «береговую линию» представлен в этой статье на Хабре, наш же вариант описан выше, и больше похож на тот, который предложен [здесь](#).

Обработка события круга

Событие круга — это когда мы вытащили из очереди CircleEvent.

Что в нём есть? В нём есть точка — это самая нижняя точка некоторой окружности, которая проходит через три каких-то сайта, и арка, которую следует удалить. В дереве есть две её контрольные точки и она сама, контрольные точки в итоге превратятся в одну, а арку нужно аккуратно удалить из дерева, перестроив все «связи родителей-детей». По сути, обработка этого события заменяет в дереве три узла на один (два break point `a и арку на один break point).

Стоит отметить, что событие круга завершает два ребра диаграммы Вороного, то есть при обработке этого события рёбра будут завершаться.

Также скажу, что важной частью обработки событий является то, как мы следим за ростом рёбер, добавлением их в список и завершением рёбер, то есть за тем, чтобы в итоге они все были «закончены», и либо были конечны, либо упирались в ограничивающий прямоугольник.

Небольшой анализ результата

Так как мы получим на выходе уже РСДС (DCEL), то сможем получить достаточно информации — для каждого ребра мы знаем соответствующий сайт, можем получить «близнеца» этого ребра, узнать сайт для него и вуаля — мы узнали соседа (так, пройдя по всему списку, можем сформировать списки соседей для всех сайтов, а это уже достижение, так как всё произошло в итоге за $O(n * \log(n)) + O(n) = O(n * \log(n))$).

Причём если у ребра «нулевой» сайт, то мы оказались на «граничном сайте» — сайте с «бесконечным» локусом. Хмм, но ведь это позволит нам построить выпуклую оболочку начального множества точек за $O(n)$ в итоге, здорово.

Ну и вообще, РСДС — это, по сути, граф, так что можно продолжать работать с этим списком во многих алгоритмах на графах.

Рекурсивный алгоритм построения диаграммы Вороного за $O(n * \log(n))$

Данный алгоритм приводится в книге [1] (стр. 260), здесь я приведу только сам алгоритм построения, поскольку данный вариант мы не реализовывали, хотя он является хорошей аналогией алгоритму Форчуна.

Алгоритм

1. Делим всё множество сайтов S на две примерно равные части (может быть нечётное количество точек) S_1 и S_2 ;
2. Рекурсивно строим диаграммы Вороного для S_1 и S_2 ;
3. Объединить полученные диаграммы и получить диаграмму для S .

Общее описание алгоритма не сложное, однако в алгоритме есть свои тонкости, с которыми Вы можете ознакомиться в указанной книге.

Применения

Исчерпывающий список всех применений диаграммы Вороного находится [здесь](#), я же упомяну некоторые из них, показавшиеся мне наиболее интересными (многие сведения взяты из [1]):

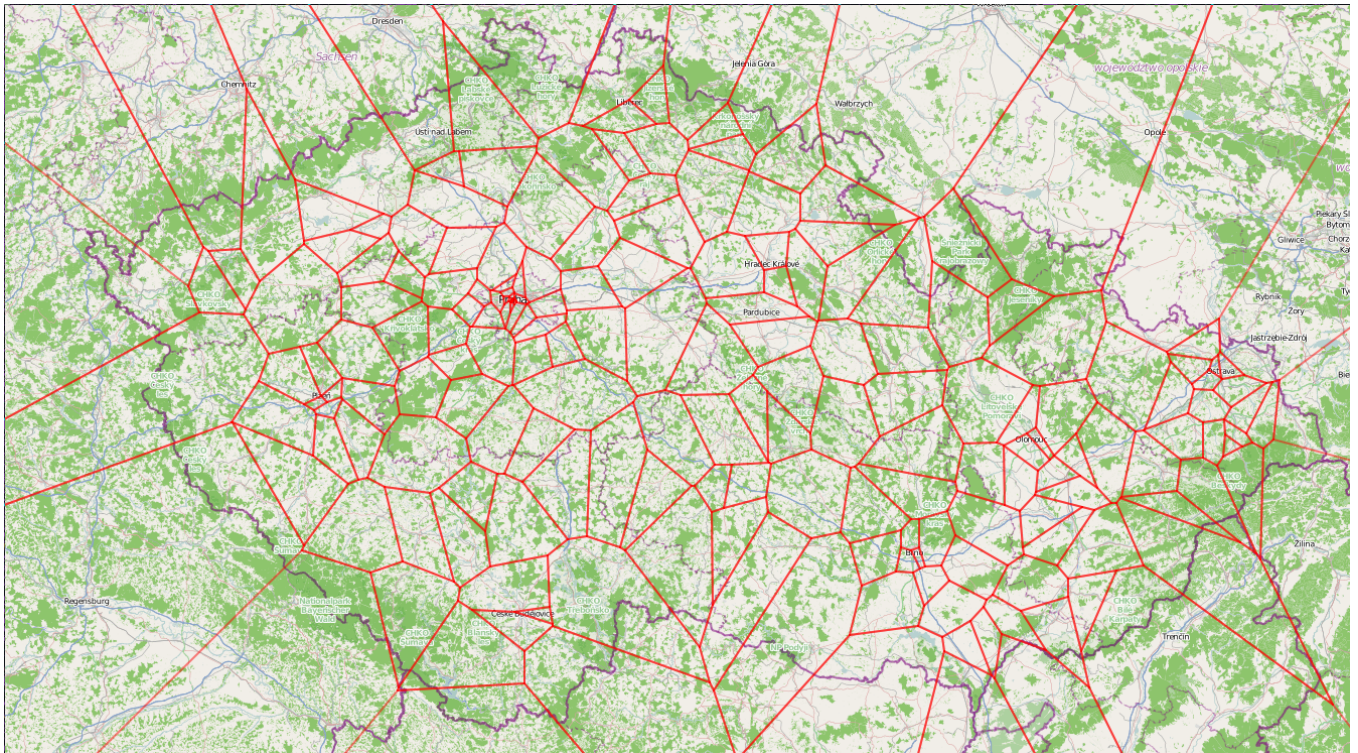
В программировании, разработке игр и картографии

В вычислительной геометрии диаграмма Вороного нужна прежде всего для решения задачи **близости** точек, а точнее, особый выигрыш диаграмма даёт в решении задачи **ВСЕ ближайшие соседи** (не те, которые громко включают музыку, а хотя...), потому как аналогичные ей способы не так просты ([1]). Используя диаграмму Вороного можно построить выпуклую оболочку за $O(n)$ (смотреть на «лучевые» рёбра, находить сайты, к которым они принадлежат, и включать их в оболочку).

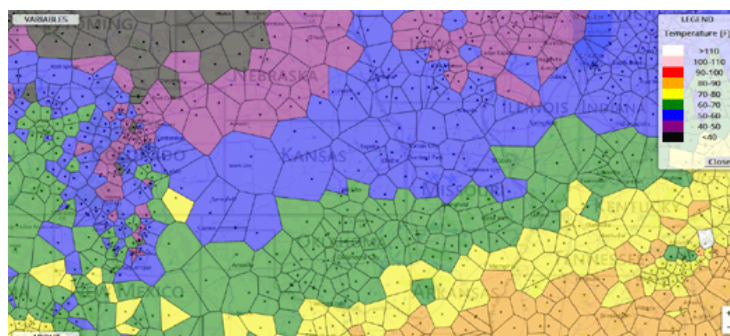
Также существует важная связь диаграммы Вороного с **триангуляцией Делоне**, которая позволяет строить одно по другому за $O(n)$, потому как они являются двойственными друг к другу (соединяем рёбрами соседние сайты, в итоге получим триангуляцию Делоне — [викиконспекты](#)).

Пример использования диаграммы Вороного в геймдеве можно найти, например, [в этой статье](#) — здесь система навигации в игровом движке основана на диаграмме.

Не исключено, и даже вероятно, что различный геолокационный софт использует диаграммы Вороного. Геолокационные рекомендательные системы могут использовать диаграмму Вороного для определения, например, ближайшего к вам продуктового магазина, для различного поиска и анализа местоположения.



Здесь же можно упомянуть и применение диаграммы в картографии — для очерчивания границ регионов и дальнейшего анализа на их основе. Да и вообще, любые географические диаграммы, показывающие распределение чего либо, можно наглядно проиллюстрировать с помощью раскрашенных диаграмм Вороного, и там будет виден переход нужного нам показателя (например, температуры):



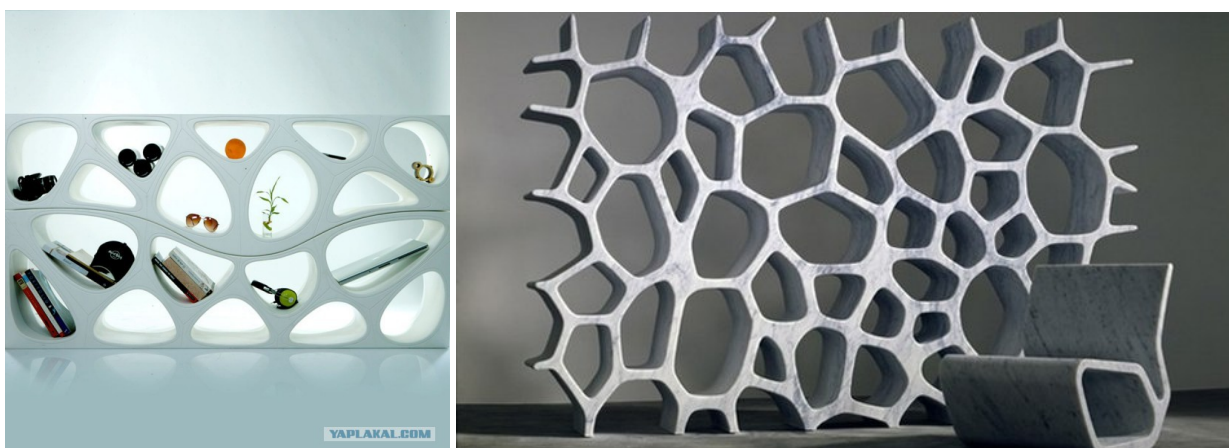
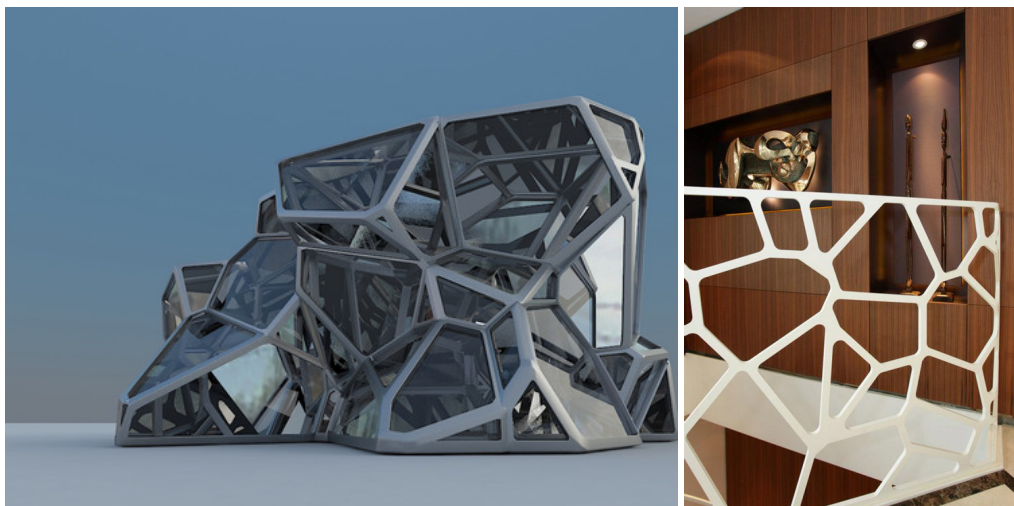
Ну и, конечно, можно делать различные фильтры-обработчики фото с помощью диаграммы Вороного, получая некую «мозаику».



Но это только начало её применения.

В архитектуре и дизайне

Весьма логично, что людям в голову пришла идея использовать диаграмму Вороного в архитектуре и дизайне, поскольку она сама по себе является красивым рисунком, своего рода «геометрической паутиной», так что есть много случаев применения её в качестве одного из основных элементов композиции или даже каркаса всего творения. Примеры:



В археологии

Из [1]:

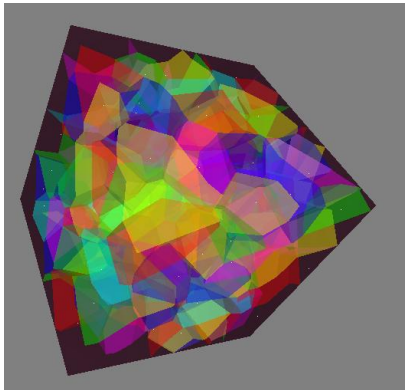
В археологии многоугольники Вороного используются для нанесения на карту ареала применения орудия труда в древних культурах и для изучения влияния соперничающих центров торговли.

В экологии возможности организма на выживание зависят от числа соседей, с которыми он должен бороться за пищу и свет.

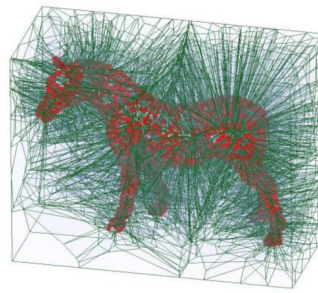
— что вполне логично, ведь обычно за любое «выживание» бьются именно соседствующие регионы.

В моделировании и распознавании

В этой статье 3D-диаграмма Вороного не рассматривается, однако она имеет многие приложения в физике и 3D-моделировании объектов. Разного рода сетки (и *скелеты*) объектов в пространстве можно построить с помощью диаграммы Вороного (однако чаще с помощью **триангуляции Делоне**).



(a)



(b)

3D-сканирование (и [компьютерное зрение \(computer vision\)](#)) различных объектов тоже может использовать диаграмму Вороного и триангуляцию Делоне, также это тесно связано с *робототехникой* — движение робота с учётом препятствий на пути.



В биологии и химии

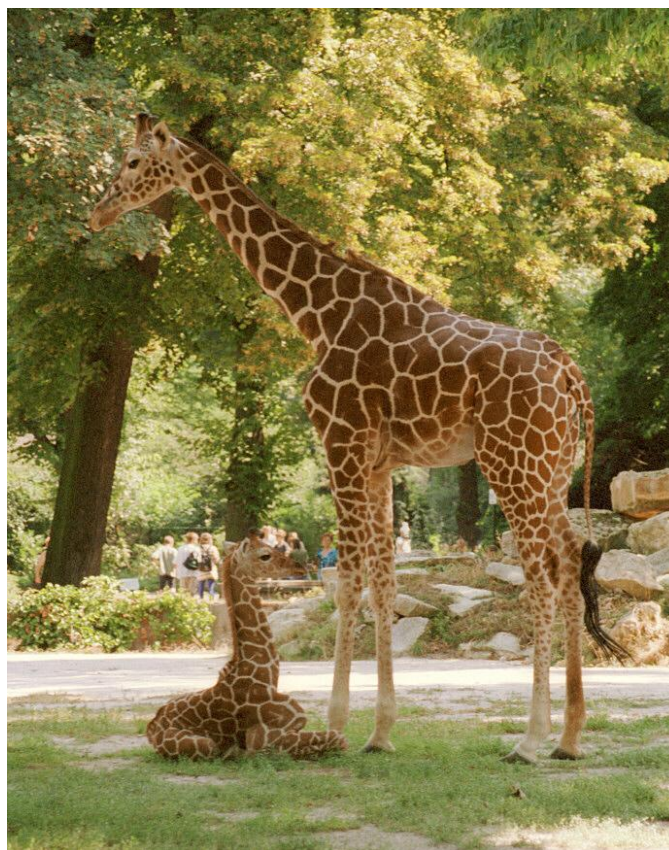
Из [1]:

Совместное влияние электрических и близкодействующих сил, для изучения которых строятся сложные диаграммы Вороного, помогает определять структуру молекул.

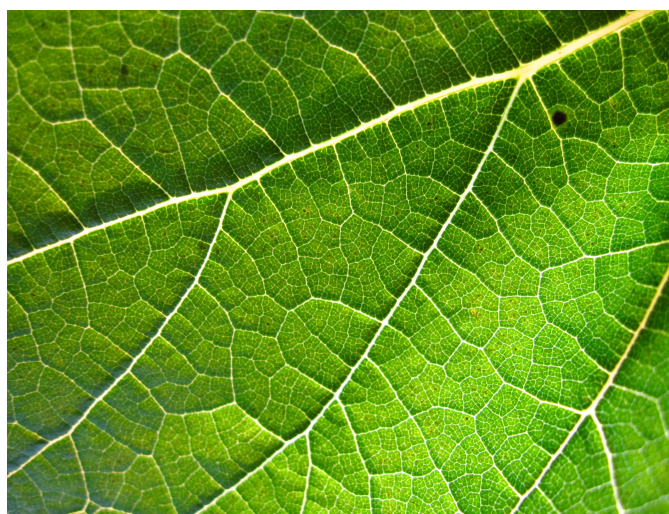
Вот ещё одна [интересная статья](#) про применение диаграммы Вороного.

Интересные факты

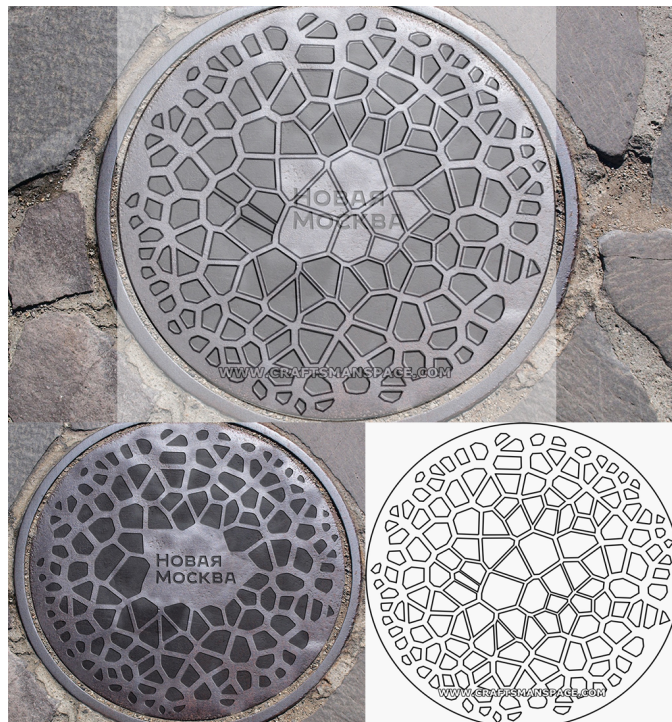
Природа — удивительная вещь, ведь оказывается, что **окрас жирафа** фактически имеет вид диаграммы Вороного. Это видно невооружённым глазом:



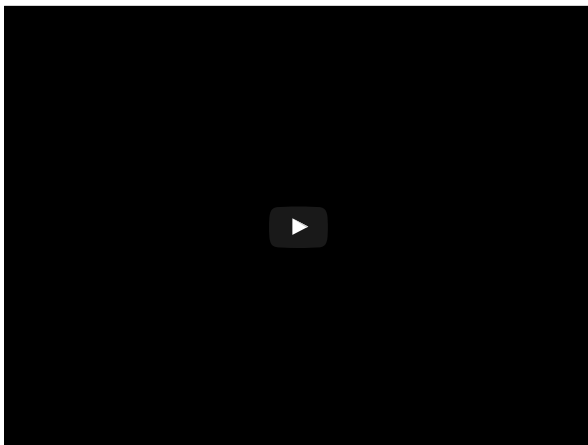
Также примечательно и следующее наблюдение, которое показывает, что диаграмму можно увидеть даже на листьях деревьев:



Кстати, чуть меньше, чем год назад, с диаграммой Вороного также был связан один инцидент — здесь она использовалась в логотипе проекта «Новая Москва».



И — напоследок — видео, в котором видно, как диаграмма появляется при росте окружностей с центрами в сайтах:



Здесь можно провести аналогию с распространением пожара, имеющего несколько очагов возгорания.

Ну, вот и подошло к концу наше небольшое путешествие в мир, где все знают, кто находится к ним ближе всего. Надеюсь, вам понравилась эта статья и вы действительно узнали что-то новое, полезное и интересное.

Спасибо за внимание!

Список литературы

- [1] Препарата Ф., Шеймос М. Вычислительная геометрия. Введение (1989)
- [2] Александров А. Д., Вернер А. Л., Рыжик В. И. Стереометрия. Геометрия в пространстве
- [3] Joseph O'Rourke. Computational Geometry in C

» Статью подготовил студент 1-го курса ФИВТ МФТИ **Захаркин Илья**.

» В написании библиотеки также помогли студентки 1-го курса ФИВТ МФТИ **Кириленко Елена** и **Касимова Надежда**.


» В тестировании библиотеки помогал **Ярослав Спирин**.

» Отдельная благодарность менторам **Гадельшину Ильнуру** и **Гафарову Рустаму**.

📌 алгоритмы, вычислительная геометрия, геометрия, диаграмма Вороного, пересечение прямых, пересечение парабол, пересечение отрезков, пересечение многоугольников, алгоритм Форчуна, береговая линия, заматающая прямая

↑ +86 ↓
👁 24,2k
★ 228

🐦
👤
📘
❤



Илья Захаркин @QuantZero

Пользователь

карма рейтинг

23,0 0,0

Похожие публикации

- +16

Обсуждение работы алгоритма Романова на примере

1,1k

19

24
- +44

Открытое письмо ученым и эталонная реализация алгоритма Романова для NP-полной задачи 3-Вып

5,1k

32

132
- +145

Алгоритм «diamond-square» для построения фрактальных ландшафтов

61,4k

608

54



Спецпроект

Самое читаемое

Разработка

Сейчас Сутки Неделя Месяц

- +13

Одна простенькая задачка. Быстро, красиво или чисто?

9,7k

59

27
- +19

Парадокс Rimworld: захватывающая сюжетом «песочница»

2,3k

18

8
- +11

Объясняем бабушке: Как зашифроваться за час

4,3k

60

21
- +25

Log in или Log on? Front-end или Frontend? Продолжаем разбираться

2,3k

25

16
- +10

Как я добавил 30000 человек в первый круг контактов, а эту соцсеть заблокируют в РФ

2,1k

11

6

Комментарии (47)

- mayorovp

5 сентября 2016 в 19:23

#

+3

↑

↓

> Есть теорема, в которой говорится, что вершина диаграммы Вороного всегда лежит на пересечении ровно трёх рёбер диаграммы

Контрпример: постройте диаграмму Воронова для квадрата

Semenar

5 сентября 2016 в 19:48

#

h

↑

+2

↑

↓

Для набора точек, из которых никакие четыре не лежат на одной окружности это верно. Если мы накидаем случайных точек, то так будет практически всегда (малым шевелением точек исправляется любой контрпример).

Deosis

6 сентября 2016 в 09:32

#

h

↑

+5

↑

↓

https://habrahabr.ru/post/309252/

20/25

Так можно доказать, что случайное натуральное число является нечётным (малым изменением числа можно исправить любой контрпример)



Semenar 6 сентября 2016 в 09:36 # h ↑

0 ↑ ↓

Ну так в доказательстве нечётности мы сильно ограничены в том, какие малые шевеления делать, тут же — практически любые (площадь «запрещённого куска» равна нулю).



DistortNeo 6 сентября 2016 в 09:43 # h ↑

0 ↑ ↓

А теперь представьте себе, что шевелить точки мы не можем. Например, когда точки — пиксели.



Semenar 6 сентября 2016 в 09:46 # h ↑

-1 ↑ ↓

Ну так статья не про этот случай, вы сами ниже сообщаете, что если точки — пиксели, то называется это уже по-другому.



DistortNeo 6 сентября 2016 в 09:56 # h ↑

0 ↑ ↓

А разве общие алгоритмы нельзя применять к изображениям?

Даже если не изображения, сетка часто бывает дискретной. А когда результат работы алгоритма зависит от рандома — это ну очень странно.



Semenar 6 сентября 2016 в 10:01 # h ↑

-1 ↑ ↓

Для той теоремы, с которой началось обсуждение, вряд ли рассматривалась сетка. В реальности из-за более строгих ограничений на расположение точек (в узлах сетки) она может и не выполняться.

А то, что я про случайные точки написал — это её доказательство, по сути.



mayorovp 6 сентября 2016 в 11:02 # h ↑

0 ↑ ↓

Не путайте доказательство и условие применимости.



Semenar 6 сентября 2016 в 13:05 # h ↑

-2 ↑ ↓

Мы берём какой-то набор точек и рассматриваем теорему для него. Для практически всех наборов она оказывается верна. Это не означает, что её можно применять только к случайно набросанным точкам, просто для точек на сетке она оказывается немного неверна (число вариантов расстановки конечно, из них есть не удовлетворяющие условию — те же точки в вершинах квадрата).



Semenar 6 сентября 2016 в 13:06 # h ↑

-1 ↑ ↓

Если сетка сама ограничена, да.



DistortNeo 6 сентября 2016 в 13:40 # h ↑

+1 ↑ ↓

Теорема не может быть «немного неверна», математика — точная и строгая наука. Теорема либо верна, либо нет. В данном случае теорема неверна.

Можно наложить условие «никакие 4 точки не должны лежать на одной окружности». Тогда теорема будет справедлива. Но это условие избыточно.

Реально полезная информация — это выпуклость полученных кластеров.



mayorovp 6 сентября 2016 в 13:40 # h ↑

+1 ↑ ↓

Математическая теорема не бывает «немного неверна». У нее есть вполне конкретное условие — если никакие 4 точки не лежат на одной окружности.

Для случайных точек на плоскости можно использовать формулировку «вершины диаграммы имеют степень не более 3 с вероятностью 1».

Дискретная сетка с произвольными точками ни к одному из этих вариантов условия не подходит (там есть точки на одной окружности и используется не вся плоскость).



ParaBubaDiop 5 сентября 2016 в 19:49 # h ↑

+4 ↑ ↓

Еще более контрпример — постройте диаграмму для произвольно наброшенных точек на окружности.



Meklon 5 сентября 2016 в 19:23 (комментарий был изменён) #

+2 ↑ ↓

Спасибо за публикацию. С удовольствием прочитал. Еще частое применение — биоинформатика. Анализ распределения клеток и структур на фотографии, определение соседей и регионов.



DaylightIsBurning 5 сентября 2016 в 19:49 # h ↑

+2 ↑ ↓

Добавлю, что алгоритм применяется в флуоресцентной микроскопии (фотографии клеток) для определения границ клеточной мембраны. За центры берут клеточные ядра. Применяются диаграммы Вороного, правда, не потому что очень хорошо работают, а потому что это — простой алгоритм. Но на самом деле клеточные мембраны не ведут себя в соответствии с алгоритмом Вороного и в некоторых случаях различия столь значительны, что приходится вручную указывать границы клеток.

 **citirex** 5 сентября 2016 в 19:53 #

+1 ↑ ↓

Спасибо за статью.

Вспомнил, как первый раз реализовал алгоритм Форчуна, тогда это было для меня достижением и улыбка не слезала с лица очень долго.

 **EndUser** 5 сентября 2016 в 21:49 # h ↑

0 ↑ ↓

Аналогично, для определения присвоенных участков сочинял алгоритм раздувающихся кругов. Был самодоволен. О том, что это диаграмма Вороного узнал значительно позже.

 **pehat** 5 сентября 2016 в 19:57 #

+3 ↑ ↓

Невооруженным глазом нашел на теле жирафа невыпуклое пятно. Не надо приплетать, пожалуйста.

 **QuantZero** 5 сентября 2016 в 20:33 # h ↑

-1 ↑ ↓

Вы правы, у жирафов рисунок не всегда в точности является диаграммой, тем более, что у всех жирафов он разный. Однако в тексте отмечено, что он **фактически** является диаграммой, поскольку сходство очевидно, и наличие нескольких линий, делающих многоугольники рисунка невыпуклыми, этого факта не отменяют.

 **pehat** 5 сентября 2016 в 23:01 # h ↑


+11 ↑ ↓

Забавно, что Вы опровергаете наблюдаемый факт словом «фактически».

 **sergeypid** 6 сентября 2016 в 09:17 # h ↑


+3 ↑ ↓

Пятна на коже жирафа и других животных более корректно моделируются клеточными автоматами.

 **Shortki** 6 сентября 2016 в 01:44 # h ↑

+1 ↑ ↓


Если присмотреться к фото, то видно что ранее такие пятна были разбиты на выпуклые, но границы между ними стали очень тонкие и теперь почти не видны. Похоже пигментация кожи начинается с неких случайных зёрен которые растут пока не встретятся на границе.

 **Halt** 6 сентября 2016 в 07:59 (комментарий был изменён) # h ↑

+2 ↑ ↓

Отличное видео на эту тему с канала MinuteEarth:

<https://youtu.be/aIH3yc6tX98>

 **beTrue** 5 сентября 2016 в 22:30 (комментарий был изменён) #

+1 ↑ ↓

Автор, отличная статья, большое спасибо!

Ссылка на реализацию 3d на JS с помощью WebGL <http://wwwmpa.mpa-garching.mpg.de/~dnelson/webgl/vormesh3.htm>, в свое время пришлось долго разбираться, данная статья будет отличным пособием.

 **slik** 5 сентября 2016 в 22:42 #

+3 ↑ ↓

Как пример использования — [процедурная генерация карты](#)

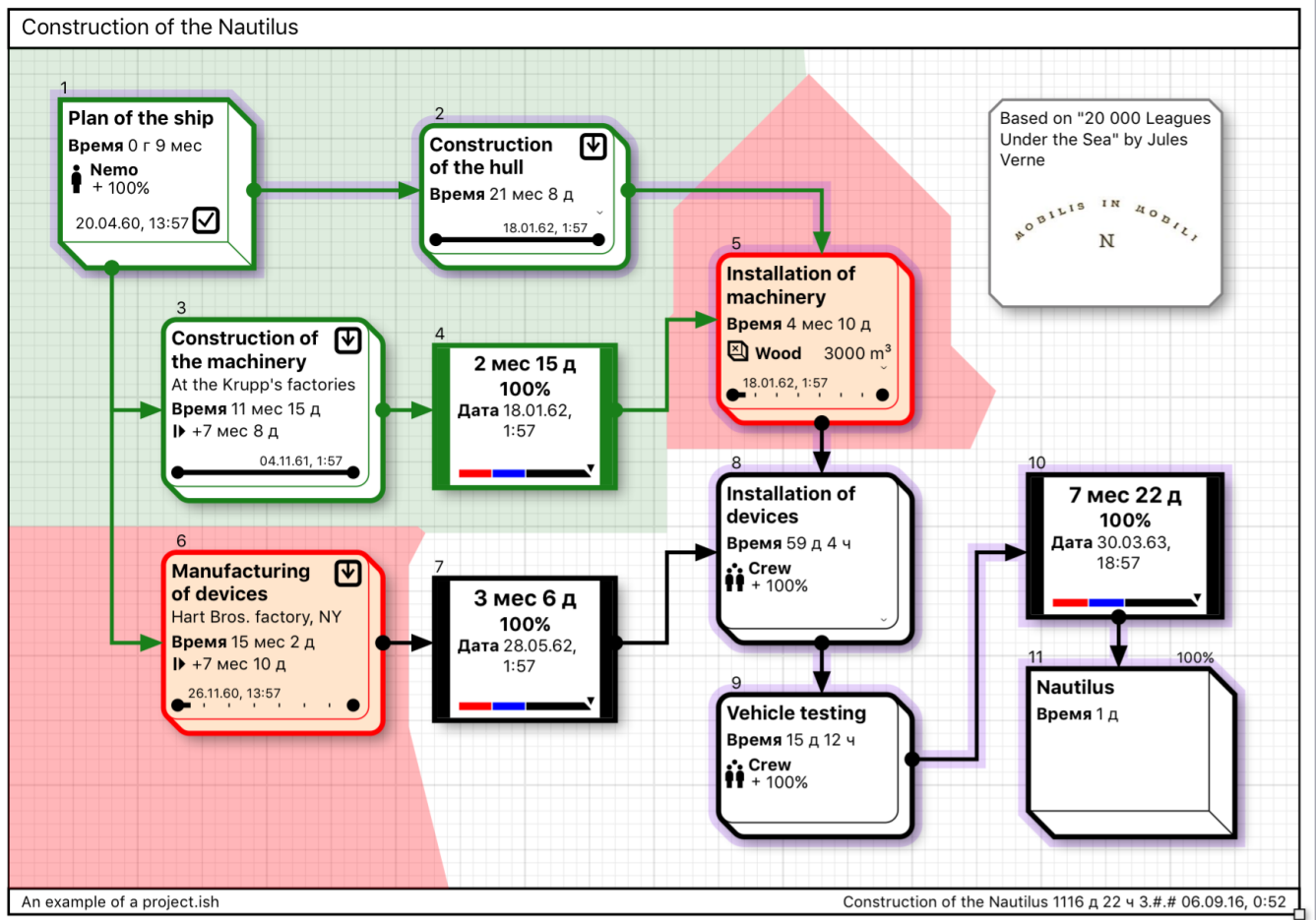
 **Shortki** 6 сентября 2016 в 01:29 #

0 ↑ ↓

У меня в приложении возникла похожая задача: для диаграммы с прямоугольными объектами в разном состоянии нужно найти разбиение на области примыкающие к прямоугольнику. То есть как диаграмма Вороного, но с прямоугольниками вместо точек. Точное решение я заменил упрощённым алгоритмом, получилось удачно, с артефактами только в особых на критичных случаях, но быстро, так что можно разбиение перестраивать по мере редактирования схемы.

Теперь на диаграмме процессов у меня есть тактическая раскраска показывающая какая часть диаграммы исполнена (зелёная), какая активна

(красная), а какая пассивна (без окраски) вполне удобно и на схеме эффектно смотрится.



Drevnir 6 сентября 2016 в 01:30 #

Если, вдруг, кому надо, могу дать рабочий проект WInForms

DistortNeo 6 сентября 2016 в 09:05 #

Сразу видно, что статью писал математик — рассматриваются только непрерывные случаи.

Но не менее важно упомянуть и дискретный случай, когда вершины могут находиться только в узлах непрерывной сетки, например — пиксели изображений. В этом случае операция называется «Euclidean Distance Transform» (EDT).

При этом есть возможность построить алгоритм с меньшей сложностью — $O(N)$ вместо $O(N \cdot \log N)$ для общего случая. Кому интересно — вот [статья](#). Алгоритмы там совсем несложные — до алгоритма Meijster я додумался сам. Также существуют методы приближённого вычисления EDT, ориентированные на GPU.

Примеры применений EDT — быстрое вычисление операций математической морфологии с круговым структурным элементом произвольного радиуса, скелетизация.

dubroffsky 6 сентября 2016 в 11:48 #

FYI, для ознакомления <http://www.cgal.org/>, конкретно по диаграммам Вороного <http://doc.cgal.org/latest/Manual/packages.html#PartVoronoiDiagrams>

wataru 6 сентября 2016 в 13:41 #

Есть (спорно) более простой метод построения диаграммы за $O(N^2 \log N)$. Все, почти как у вас, только полуплоскости пересекаются по-другому. Не надо пересекать выпуклые многоугольники, можно так и работать с полуплоскостями.

Сначала их надо отсортировать по углу наклона (при чем ориентировать полуплоскости так, чтобы центр локуса был слева от прямой). В этом случае сортируем против часовой стрелки. Потом добавляем плоскости по одной, отсекая от текущей границы ненужные отрезки. Очень удобно текущую границу представить в виде стека отрезков — первый не замкнут слева, остальные замкнутые, но последний не замкнут справа. Когда добавляем новую прямую, пересекаем ее с отрезком вершиной стека. Если пересечение левее левой границы отрезка, просто выбрасываем его и повторяем операцию. В конце мы получим этакую петлю образованную прямыми. Надо будет ее обрезать с двух концов — это тоже просто. Пересекаем 2 прямые, первую и последнюю в стеке. Если точка пересечения левее левого конца на последней, то выкидываем ее из стека. Иначе если точка пересечения правее правого конца на первой прямой, выкидываем ее. В противном случае замыкаем точкой пересечения первую и последнюю прямые. Вот мы и получили весь локус.

Нужно только уметь пересекать прямые параметрически, и сортировать их по углу. Отрезки хранятся в виде прямой и допустимого отрезка параметров на ней. В самом конце надо выбрасывать элементы с начала стека, что по идее требует дека, но если реализовать стек массивом, то можно просто двигать указатель на начало и потом в конце уже сдвинуть все элементы в начало.



zemen96 6 сентября 2016 в 16:51 # h ↑

+1 ↑ ↓

Есть еще более простой метод пересечь полуплоскости. Если сделать полярное преобразование прямых, образующих полуплоскости, относительно точки внутри пересечения, получится N точек. Далее можно построить их выпуклую оболочку и сделать полярное преобразование отрезков оболочки. Оно и будет вершинами искомого пересечения.



wataru 6 сентября 2016 в 17:15 # h ↑

0 ↑ ↓

Жесть какая это ваше полярное преобразование! Есть ли у вас хорошие ссылки, где про это прочитать?

Обычная инверсия, это еще понятно, но тут точке сопоставляется прямая и наоборот. И главное, совсем-совсем непонятно, почему прямая через образы двух прямых соответствует образу точки пересечения. И почему выпуклая оболочка будет внутренней границей всех пересечений тоже непонятно в таком преобразовании.



wataru 6 сентября 2016 в 17:33 # h ↑

0 ↑ ↓

Блин, вы абсолютно правы! Полярное преобразование — просто гениальная вещь! У вас нет ли примеров хороших задачек, где это преобразование еще можно применить?



garan 6 сентября 2016 в 16:52 #

+2 ↑ ↓

Локус, сайт... Потому что пудинг — очень вкусный блюдинг...

Читайте первоисточники и учите термины на русском языке. Сама работа Вороного 2008г. не очень поможет, она на французском), но есть обзор работ Вороного, который сделал Борис Николаевич Делоне в книге «Петербургская школа теории чисел», 1947г. Еще есть работа Делоне, Сандаковой по теории стереоэдров. Там используется правильная терминология. Также очень советую разобраться с историей вопроса «подъема точек на параболоид» (lifting). А также историю вопроса «метода оборачивания подарка» (gift wrapping). Кто и сколько раз это переоткрывал. А придумал Вороной в 1908г.

А в целом статья хорошая.



Saso 6 сентября 2016 в 16:52 #

0 ↑ ↓

Меня удивило, что в библиотеке SplashGeom 2D-линия задается всего тремя числами. Я всегда считал, что необходимо четыре. Может кто-нибудь пояснить, за счет чего достигнута такая «экономия».



DistortNeo 6 сентября 2016 в 16:59 # h ↑

+3 ↑ ↓

Четыре числа нужно для записи отрезка в 2D. А вот прямая параметризуется всего двумя числами (a, d):

$$x \cos a + y \sin a = d$$

Но так как работать с синусами и косинусами неудобно, то представляют в виде:

$$Ax + By = C$$

и иногда нормализуют ($A^2 + B^2 = 1$).



Saso 7 сентября 2016 в 11:36 # h ↑

0 ↑ ↓

Спасибо. Исчерпывающий ответ.



wataru 6 сентября 2016 в 17:02 # h ↑

0 ↑ ↓

Просто уравнение прямой $Ax + By + C = 0$. Но тут есть одна степень свободы, на самом деле достаточно 2х чисел ($y = kx + b$, или $\cos(a)x + \sin(a)y + c = 0$). 4 числа нужно, если задавать прямую 2-мя точками, но при этом тут 2 степени свободы.



DistortNeo 6 сентября 2016 в 17:12 # h ↑

+3 ↑ ↓

Вариант $y = kx + b$ обычно не используют из-за невозможности представления вертикальных прямых. Смысл в добавлении степени свободы — избавиться от дорогостоящих операций (\sin , \cos , $\sqrt{}$).



LynXzp 6 сентября 2016 в 22:36 (комментарий был изменён) # h ↑

0 ↑ ↓

И тут я замер... вспоминая во скольких программах у меня используется такое уравнение прямой.



DaylightIsBurning 7 сентября 2016 в 16:20 # h ↑

-1 ↑ ↓

можно использовать $y = \tan(a)x + b$



mayorgovp 7 сентября 2016 в 17:03 (комментарий был изменён) # h ↑

0 ↑ ↓

Нельзя



Westimo 7 сентября 2016 в 18:13 #

+2 ↑ ↓

В свое время (года 2-3 назад) тоже занимался построением этой диаграммы, а потом и триангуляцией по этой диаграмме. С работой этих алгоритмов в 2D все понятно и достаточно просто представить. Но когда пытаешься перейти в 3D начинается взрыв мозга.

Причем в большинстве материалов первыми идут строки, вроде: «будем рассматривать 2-мерное пространство, для 3-мерного все аналогично».

Было бы очень интересно прочесть про построение диаграммы в 3D, ND с визуализацией



QuantZero 7 сентября 2016 в 19:33 # h ↑

0 ↑ ↓

Согласен, в 3D всё будет уже не так интуитивно понятно и просто. В будущем, возможно, я напишу статью и на эту тему (однако сначала реализация алгоритма Форчуна).



DistortNeo 7 сентября 2016 в 19:46 # h ↑

0 ↑ ↓

При переходе в 3D используется сведение задачи к 2D с помощью теоремы Пифагора.

Алгоритм такой:

1. Применяем обычный 2D алгоритм к каждой из плоскостей XY.
 2. Далее для каждой из точек XY рассматриваем прямую, перпендикулярную XY.
 3. Вращение ближайших точек, найденных на 1 шаге, не влияет на результат => вращаем так, чтобы они все оказались в одной плоскости.
 4. Применяем 2D алгоритм.
- И так для каждой последующей размерности.



iroln 7 сентября 2016 в 18:14 (комментарий был изменён) #

+1 ↑ ↓

Использовал диаграмму Вороного и теорию графов для вычисления Medial axis (центральной линии) 2D-объектов (для 3D объектов, заданных сеткой, тоже можно). По сути — это построение скелета. Кратчайший путь через диаграмму Вороного для 2D-случая как раз даёт центральную линию. Правда пришлось немного помудрить в случае самопересечений объекта.

► [Картинки](#)

Только зарегистрированные пользователи могут оставлять комментарии. [Войдите](#), пожалуйста.

Интересные публикации



Новые-старые форматы: HD-винил и DIY-пластинки 1



Делаем стартап просто и технологично. Маячки Eddystone 0



Производительность межпроцессного обмена сообщениями в node.js 0



SMART TV – будущее телевидения 5



Выпуск Rust 1.13 4