МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования

«Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского» (ННГУ)

Институт информационных технологий, математики и механики

ОТЧЕТ

по практике

«Идентификация функции конкурентоспособности методами машинного обучения на основе динамики спроса»

Выполнил: студент группы 382006-1

Мусин А.Н _____

Проверил: доцент, к.ф.-м.н.

Кузенков О.А._____

Нижний Новгород

2024

Содержание

1 Введение **3**

2 Постановка задачи **4**

3 Экономические подходы в оценки конкурентоспособности товаров **5**
3.1 Дифференциальный метод оценки конкурентоспости 5
3.2 Комплексный метод оценки конкурентоспособности 7

4 Методика расчета конкурентоспособности **9**

5 Реализация **12**
5.1 Сбор данных 12

6 Востановление функции конкурентоспособности с помощью линейных параметров. **16**

7 Приложение **17**
7.1 Исходный код c++ 17
7.1.1 Последовательная реализация 17
7.1.2 OMP 18
7.1.3 TBB 20
7.1.4 std::thread 22
7.2 Тесты последовательной реализации 23
7.3 Тесты параллельных реализаций (отличаются друг от друга функцией get_convex_hull_[название технологии]) 25

1 Введение

Товар является центральным объектом на рынке, обладающим определенной потребительской ценностью, качеством, техническим уровнем, надежностью и прочими важными характеристиками. Несмотря на разнообразие аналогов, покупатель должен выбрать продукт, который лучше всего соответствует его потребностям, учитывая все характеристики. Конкуренция, фундаментальное понятие в современной экономике, представляет собой борьбу между товарами за право существования на рынке, стимулируя производителей к улучшению продукции для удовлетворения потребностей клиентов.

Конкурентоспособность определяется способностью товара превзойти аналоги по техническим, экономическим и эксплуатационным характеристикам, что поддерживает его на рынке. Это сравнение с конкурентами выявляет наилучший и наихудший варианты с точки зрения потребительских предпочтений. Анализ рынка и оценка конкурентоспособности товаров становятся ключевыми задачами для производителя, который может использовать различные методики, включая машинное обучение, для выявления преимуществ и недостатков своей продукции.

Методы машинного обучения эффективны для анализа объемных данных, выявления тенденций и прогнозирования в условиях динамичного рынка. Предложенный в работе подход к использованию машинного обучения для прогнозирования конкурентоспособности товара и выявления предпочтений покупателей подчеркивает ограниченность современного экономического подхода, основанного на линейной зависимости параметров товара.

2 Постановка задачи

Цель данного исследования заключается в использовании методов машинного обучения для выявления функции конкурентоспособности на основе динамики спроса. Для достижения этой основной цели мы начнем с анализа существующих экономических методик оценки конкурентоспособности, близких к теме исследования, с целью выявления их преимуществ и недостатков.

Затем мы разработаем и обоснуем новый метод расчета функции конкурентоспособности, основанный на применении методов машинного обучения, ранжировании и анализе динамики спроса.

Применяя наш метод к немецкому рынку центральных процессоров за период с августа 2020 по апрель 2022 года, мы восстановим весовые коэффициенты, чтобы выявить наиболее и наименее значимые характеристики товара.

Затем определим значения функции конкурентоспособности для каждой модели, чтобы оценить потенциальные позиции на рынке. В ходе исследования мы продемонстрируем, что невозможно восстановить функцию конкурентоспособности как линейную комбинацию.

Наконец, мы решим задачу подбора оптимальных значений характеристик для "начинающего"производителя с целью успешного введения нового процессора на рынок.

3 Экономические подходы в оценки конкурентоспособности товаров

В настоящее время применяются различные методики оценки конкурентоспособности, которые успешно используются в практике предприятий и товаров. Каждая из этих методик имеет свои преимущества в различных сценариях, однако они также обладают существенными недостатками, которые могут привести к ошибочным результатам в данной задаче.

В связи с этим необходимо провести анализ существующих методик, выявив их сильные и слабые стороны. Только после этого можно предложить собственный метод, учитывающий выявленные недостатки.

Одной из методик, используемой в оценке конкурентоспособности, является дифференциальный метод. Его суть заключается в анализе отдельных (технических и экономических) параметров рассматриваемой продукции и их сопоставлении с потребностями. При использовании этого метода можно определить, достигла ли продукция необходимого уровня по всем параметрам, выявить параметры, которые не соответствуют требованиям, и выделить те из них, которые сильнее всего отличаются от базовых.

3.1 Дифференциальный метод оценки конкурентоспости

1. Если в качестве отправной точки для оценки используются стандартные показатели качества товаров, то расчет единичного показателя конкурентоспособности выполняется с применением следующей формулы:

$$q_{ni} = \begin{cases} 0, & \text{если } P_i < P_{i0}, \\ 1, & \text{если } P_i \geq P_{i0} \quad i = 1, 2, \ldots, n \end{cases}$$

Где q_{ni} – единичный параметрический показатель конкурентоспособности по i-му нормативному параметру;

P_i – величина i-го параметра для анализируемой продукции;

P_{i0} – величина i-го параметра для изделия, принятого за норму;

n – количество параметров;

При использовании нормативных параметров единичный показатель оценки может быть всего лишь двух видов – 1 или 0. В случае соответствия анализируемой продукции установленным нормам и стандартам,

показатель принимает значение 1, в то время как при отклонении параметров продукции от установленных норм и стандартов, показатель становится равным 0.

2. Если мы принимаем степень удовлетворения потребностей потребителя в качестве базового критерия для оценки конкурентоспособности товаров, то расчет единичного показателя конкурентоспособности производится с использованием следующей формулы:

$$q_i = P_i/P_{i0} * 100\% i = 1, 2, \ldots, n$$

$$q_i' = P_{i0}/P_i * 100\% i = 1, 2, \ldots, n$$

Где q_i, q_i' — единичный параметрический показатель конкурентоспособности по i-му нормативному параметру;

P_i – величина i-го параметра для анализируемой продукции;

P_{i0} – величина i-го параметра для изделия, принятого за норму;

n – количество параметров;

Из предложенных формул выбирается та, в которой увеличению единичного показателя соответствует повышение уровня конкурентоспособности.В случае, если характеристики исследуемого товара не могут быть измерены количественно (например, параметры вкуса, цвета, консистенции, запаха), применяются экспертные методы оценки в баллах. В этом случае подвергают оценке в баллах как исследуемый образец, так и базовый.

Дифференциальный метод оценки конкурентоспособности ограничивается определением уровня конкурентоспособности по отдельному показателю. Он не учитывает важность каждого параметра, влияющего на выбор товара потребителем. По этой причине дифференциальные методы оценки конкурентоспособности применяются обычно в двух сценариях: при использовании степени удовлетворения потребности потребителя и при оценке соответствия нормативнотехнологическим требованиям.

3.2 Комплексный метод оценки конкурентоспособности

Комплексный метод оценки конкурентоспособности базируется на использовании различных видов показателей (групповых, обобщенных и интегральных) и сравнении относительных полезных эффектов между анализируемой продукцией и образом. В данном подходе происходит вычисление групповых показателей, учитывающих нормативные, технические и экономические параметры, после чего рассчитывается обобщенный показатель конкурентоспособности продукции относительно потребности, образца или группы образцов.

$$I_{NP} = \prod_{i=1}^{n} q_{ni}$$

Где I_{NP} — групповой показатель конкурентоспособности по нормативным параметрам;

q_{ni} — единичный показатель конкурентоспособности по i-му нормативному параметру;

n — количество параметров;

А групповые показатели по техническим и экономическим параметрам отражают уровень соответствия исследуемой продукции потребностям покупателя по всем техническим характеристикам и соотношение общих затрат потребителя на покупку и использование данного продукта по сравнению с товаром-образцом.

$$I_{TP} = \sum_{i=1}^{n} \alpha_i q_{ni}$$

$$I_{EP} = \frac{Z}{Z_0}$$

Где I_{TP} — групповой показатель конкурентоспособности по техническим параметрам;

Где I_{EP} — групповой показатель конкурентоспособности по экономическим параметрам;

q_i — единичный показатель конкурентоспособности по i-му техническому параметру;

α_i — весомость i-ого параметра в общем наборе из n технических параметров, характеризующих потребность;

n — число параметров, участвующих в оценке;

Z, Z_0 — полные затраты потребителя соответственно по оцениваемой продукции и образцу.

Интегральный показатель отражает разницу между сравниваемыми товарами в отношении потребительского эффекта, который приходится на каждую единицу затрат покупателя на их приобретение и использование.

$$K = \frac{I_{NP} \cdot I_{TP}}{I_{EP}}$$

Где K — интегральный показатель конкурентоспособности анализируемой продукции по отношению к изделию-образцу;

Соответственно если $K < 1$, то рассматриваемый товар уступает образцу по конкурентоспособности, а если $K > 1$, то превосходит. При этом, если, предположим, что $I_{NP} = 1$, то есть товар полностью соответствует стандартам по всем нормативным параметрам, а Z — величина постоянная, тогда I_{EP} — константа, и тогда выражение преобразуется в:

$$K = \frac{1}{C_0} \sum_{i=1}^{n} \alpha_i q_{ni}$$

Следовательно, интегральный показатель выражается в виде линейной функции его технических параметров. Однако такая оценка обладает рядом недостатков, например, коэффициенты α основаны на данных, полученных из социологических опросов группы лиц, и не всегда отражают объективную реальность. Это может создать проблемы при корректности построении функции конкурентоспособности. С другой стороны, подход к оценке конкурентоспособности товаров, вдохновленный работами Кузнецова О.А., Морозова А.Ю., Кузнецовой Г.В., Рябовой Е.А., Гарсина А., подчеркивает сложность с приспособляемостью живых существ. В свете этих исследований становится ясным, что линейная модель не всегда является приемлемой для построения функции приспособляемости. Иногда необходим переход к сверткам более высоких порядков. Исходя из этого, мы переходим к разработке новой методологии.

4 Методика расчета конкурентоспособности

Допустим, имеется l различных видов товаров с аналогичным предназначением, обозначенных как $v_1, v_2, \ldots, v_l$. Обозначим их множество как D. Для каждого товара v в момент времени t_k, где $k = 1, \ldots, n$, существует величина $x(v, t_k)$, представляющая количество данного товара, приобретенное потребителем в текущий момент времени t_k. Все значения $x(v, t_k)$ неотрицательны, а вектор $x(v|t) = (x(v, t_0), \ldots, x(v, t_n))$ характеризует общее состояние системы.

Объем спроса зависит от предпочтений покупателей, цен и характеристик товаров. Предположим, что цены на товары не изменяются со временем. Таким образом, объем спроса $x(v, t)$ на товар v определяется исключительно предпочтениями покупателей. Мы предполагаем, что $x(v, t)$ соответствует определенным свойствам:

1. $x(v, t) = 0$ означает отсутствие v на рынке в момент времени t;
2. $x(v, t) > 0$ означает наличие v на рынке в момент времени t;
3. $x(v, t)$ является непрерывной функцией v в D;
4. $x(v, t)$ является непрерывной функцией времени t;
5. стремление к нулю $x(v, t)$ со временем означает угасание объемов продаж;
6. $x(v, t)$ равномерно ограничена константой, т.е. $x(v, t) < C$ – ограничение на максимальное возможное число товаров на рынке.

Формально ранжирование товаров можно определить следующим образом. Модель v лучше, чем модель w, если:

$$\lim_{t \to \infty} \frac{x(w, t)}{x(v, t)} = 0$$

Это утверждение означает, что с течением времени (при $t \to \infty$) спрос на модель w вытесняется с рынка. Таким образом, мы вводим порядок ранжирования в множество D.

Предположим, что существует функционал $J(v)$, который сохраняет ранжирование моделей. Мы можем рассматривать его как функцию сравнения, то есть

$$J(v) > J(w) \Leftrightarrow v > w$$

где выражение $v >> w$ интерпретируется как "модель v превосходит модель w". Тогда предложенный функционал может быть идентифицирован как мера конкурентоспособности товара.

Математически, существование введенной функции конкурентоспособности подчиняется требованиям, соответствующим теореме Дебре о непрерывности. Для этого предполагается, что товары удовлетворяют критериям рациональности (соответствуют логике и потребностям потребителя, обладают высоким качеством, адекватной ценой и выполняют свою функцию в соответствии с целями приобретения) и непрерывности (доступны для приобретения и использования в любое удобное для клиента время, то есть всегда доступны без временных или иных местных ограничений, мешающих их приобретению). В таком случае функция J является непрерывной.

Из этой постановки следует, что товар v^*, при котором функция конкурентоспособности достигает максимума, будет оптимальным, поскольку эта модель в конечном итоге вытеснит все другие, занимая самую значительную долю рынка.

Общий объем спроса на рынке в момент t_k обозначим $u(t_k) = \sum_{i=1}^{l} x(v_i, t_k)$. Эта величина может принимать различные значения в зависимости от t_k, но при этом справедливы неравенства $0 < u_0 \leq u(t_k) \leq u_1$, где u_0, u_1 – положительные константы – минимальное и максимальное значения общего объема спроса соответственно. Величина $z(v_i, t_k) = \frac{x(v_i, t_k)}{u(t_k)}$ является удельным весом i-го товара на рынке в момент времени t_k.

Итак $z(v_i, t_k)$ удельный вес величины спроса i-го товара в момент времени t_k, который вычисляется из статистических данных спроса. Время рассматриваем дискретное равновестию месяцы.

$$\Delta z(v_i, t_0) = z(v_i, t_0 + (k+1)\Delta t) - z(v_i, t_0 + (k+1)\Delta t)$$

приряшения удельного веса величины спроса i-го товара. Как было показано характеристика спроса, которая так же является показателем конкурентоспособности вычисляется следующим способом:

$$J(v_i) =< ln(1 + \frac{\Delta z(v_i, t_k)}{z(v_i, t_k)}) >= \frac{1}{n} \sum_{i=1}^{n} ln(1 + \frac{\Delta z(v_i, t_k)}{z(v_i, t_k)})$$

Сравнивая обнаруженные значения для каждого товара, можно выяснить, который из них более конкурентоспособен.

Для каждой модели v динамика спроса $x(v, t)$ определяется различными

потребительскими, коммерческими и экономическими параметрами. Предположим, что общее количество таких параметров равно m. Тогда этот набор можно представлять в виде вектора $M(v) = (M_1(v), M_2(v), \ldots, M_m(v))$. Конкурентоспособность J является функцией многих переменных от $M(v)$, то есть $J = J(M(v))$.

Далее предположим, что функция J является бесконечно дифференцируемой по времени и достаточно гладкой функцией параметров M_i. Таким образом, мы можем аппроксимировать J с использованием разложения Тейлора вокруг некоторой точки M_0. В частности, мы можем рассмотреть приближение функции приспособленности в классе линейных или квадратичных функций путем свертки параметров. Ранее уже было сказано, что на основе данной линейная свертка не всегда является уместной, поэтому в данной работе ключевая роль отводится квадратичной:

$$J(M) = \sum_{i=1}^{m} \lambda_i M_i(v) + \sum_{i=1}^{m} \sum_{j=1}^{m} \lambda_{ij} M_i(v) M_j(v)$$

где λ_i, λ_{ij} - степени влияния каждого признака на общую конкурентоспособность, M_i - i-ая характеристика модели, $M_i M_j$ - произведение i-ой и j-ой характеристики.

В случае когда товар v лучше, чем w должно выполняться неравенство $J(v) > J(w)$, соответственно коэффициенты λ_k должны удовлетворять неравенству:

$$\sum_{i=1}^{m} \lambda_i (M_i(v) - M_i(w)) + \sum_{i=1}^{m} \sum_{j=1}^{m} \lambda_{ij} (M_i(v) M_j(v) - M_i(w) M_j(w)) > 0$$

Таким образом, каждая пара стратегий порождает аналогичное неравенство. Решая эту систему неравенств, можно оценить параметр λ, например, с применением линейного программирования. Однако размерность данной задачи может быть в общем случае чрезвычайно большой, и поэтому компоненты, входящие в неравенства, могут быть вычислены приближенно. Это может привести к небольшим погрешностям при решении, что, в свою очередь, может вызвать несовместность системы.

Для преодоления упомянутых трудностей мы прибегаем к попарному подходу ранжирования, используя машинное обучение для определения коэффициентов. Мы сопоставляем паре (v, w) точку $(M(v) - M(w))$ в многомерном пространстве параметров размерности m, n, а паре (w, v) - точку

$(M(w) - M(v))$. Таким образом, мы стремимся к тому, чтобы гиперплоскость вида

$$\sum_{i=1}^{m} \lambda_i (M_i(v) - M_i(w)) + \sum_{i=1}^{m} \sum_{j=1}^{m} \lambda_{ij} (M_i(v) M_j(v) - M_i(w) M_j(w)) = 0$$

разделяла эти точки. Рассматривая все возможные пары точек, мы получаем два множества точек в m, t-мерном пространстве, находящихся по разные стороны от разделяющей гиперплоскости.

Задача поиска параметра λ сводится к нахождению компонентов нормали гиперплоскости, разделяющей два множества точек. Эта задача сводится к бинарной классификации, которую можно легко решить с использованием специализированных методов машинного обучения. В частности, метод опорных векторов (SVM) хорошо зарекомендовал себя в таких задачах.

Метод опорных векторов (SVM) принимает входные данные для двух классов и возращает коэффициенты разделяющей гиперплоскости. Основная цель алгоритма заключается в поиске оптимальной гиперплоскости, которая эффективно разделяет данные на два класса.

Алгоритм устроен таким образом, что он выявляет точки, расположенные ближе всего к гиперплоскости разделения, и такие точки называются опорными векторами. Затем алгоритм вычисляет расстояние между опорными векторами и самой разделяющей плоскостью, которое называется зазором. Основная задача алгоритма заключается в максимизации этого зазора.

Лучшей гиперплоскостью считается та, которая максимально удалена от скопления точек каждого класса. Таким образом, алгоритм стремится найти гиперплоскость с максимальным зазором, обеспечивая максимальное расстояние между классами и, следовательно, повышая эффективность разделения.

5 Реализация

5.1 Сбор данных

В качестве динамики спроса на графические процессоры возьмем открытые данные игровой платформы Steam по популярности видеокарт среди геймеров. Роль игроков в видеоигры для рынка графических процессоров (GPU)

является значительной и многогранной. Геймеры не только являются основными потребителями высокопроизводительных GPU, но также существенно влияют на развитие технологий и тенденций в данной индустрии. Рассмотрим несколько ключевых аспектов этой роли:

1. **Спрос на производительность.** Геймеры требуют высокопроизводительных GPU для обеспечения плавного и качественного игрового опыта. Современные игры становятся все более графически насыщенными и требовательными к ресурсам, что стимулирует спрос на мощные графические карты.

2. **Обновление оборудования.** Игроки регулярно обновляют свои GPU, чтобы соответствовать новым стандартам и требованиям игр, что поддерживает постоянный спрос на новейшие модели видеокарт.

3. **Разработка новых технологий.** Для удовлетворения запросов геймеров производители GPU, такие как NVIDIA и AMD, разрабатывают новые технологии, включая трассировку лучей (ray tracing), улучшенные алгоритмы сглаживания и прочие графические улучшения.

4. **Конкуренция.** Высокий спрос на игровые GPU приводит к усиленной конкуренции между производителями, что ускоряет технологический прогресс и улучшает качество продукции.

5. **Оптимизация игр.** Производители GPU активно работают с разработчиками игр для оптимизации игр под свои графические карты, что обеспечивает лучший игровой опыт и стимулирует продажи как игр, так и GPU.

6. **Поддержка технологий.** Многие современные игры включают поддержку технологий, разработанных производителями GPU, таких как NVIDIA DLSS (Deep Learning Super Sampling) или AMD FSR (FidelityFX Super Resolution), что улучшает производительность и качество изображения.

7. **Обзоры и рекомендации.** Геймеры активно делятся отзывами и рекомендациями о графических картах, влияя на решения других покупателей. Популярные геймеры и стримеры имеют значительное влияние на рынок, продвигая определенные модели GPU.

8. **Киберспорт.** Развитие киберспорта также стимулирует спрос на высокопроизводительные GPU, так как профессиональные игроки требуют максимальной производительности для достижения успеха в соревнованиях.

9. **Продажи и доходы.** Игровой сегмент является одним из самых прибыльных для производителей GPU. Продажи игровых видеокарт составляют значительную долю доходов таких компаний, как NVIDIA и AMD.

10. **Инвестиции в R&D.** Доходы от продаж игровых GPU позволяют компаниям инвестировать в исследования и разработки, что ведет к созданию новых продуктов и улучшению существующих технологий.

Геймеры играют ключевую роль в формировании и развитии рынка GPU. Их требования и ожидания стимулируют инновации, поддерживают высокие объемы продаж и оказывают значительное влияние на стратегии и решения производителей графических процессоров.

В качестве характеристик возьмем открытую базу данных с результатами исполнения специальных тестов (бенчмарков).

Бенчмарки (от англ. "benchmark") — это тесты производительности, предназначенные для оценки и сравнения аппаратного и программного обеспечения. В контексте видеокарт, бенчмарки используются для измерения производительности графических процессоров (GPU) в различных задачах, таких как рендеринг игр, обработка графики, выполнение вычислений и т.д.

Почему бенчмарки важны для сравнения видеокарт:

1. **Объективное сравнение производительности** Бенчмарки предоставляют стандартный способ измерения производительности видеокарт, позволяя объективно сравнивать различные модели. Это особенно важно для геймеров, разработчиков и профессионалов, которые зависят от графической мощности для своей работы.

2. **Реальные сценарии использования** Хорошие бенчмарки включают тесты, имитирующие реальные условия использования, такие как запуск современных игр или выполнение профессиональных графических задач. Это позволяет пользователям понять, как видеокарта будет вести себя в условиях, близких к их реальным потребностям.

3. **Проверка стабильности и надёжности** Помимо производительности, бенчмарки могут выявить потенциальные проблемы с перегревом, стабильностью и совместимостью видеокарт. Это важно для долгосрочной надежности и безопасности системы.

4. **Сравнение по различным параметрам** Бенчмарки могут измерять производительность видеокарт по различным аспектам, таким как ча-

стота кадров (FPS) в играх, время рендеринга в графических приложениях, производительность в вычислительных задачах и т.д. Это дает комплексное понимание сильных и слабых сторон каждой видеокарты.

Примеры популярных бенчмарков для GPU
В данной работе будут фигурировать характеристики основанные на двух популярных бенчмарках:

1. **3DMark** Один из самых популярных и универсальных бенчмарков, используемый для тестирования игровых и профессиональных видеокарт.
2. **2DMark** Специализированный тест, который включает сценарии, моделирующие использование профессиональных графических приложений, таких как Adobe Illustrator или AutoCAD, для оценки производительности в 2D-графике.

Также рассматривается характеристика цены актуальная на момент сбора данных (февраль 2022г.) И метрика тепловыделения видеокатры (TDP). Однако так как все параметры должны положительно коррелировать с приспособленностью товара возьмем обратные величины. Далее везде будут использованны величины равные $1/price$ и $1/TDP$.

6 Востановление функции конкурентоспособности с помощью линейных параметров.

Для начала взглянем на матрицу корреляции наших параметров:

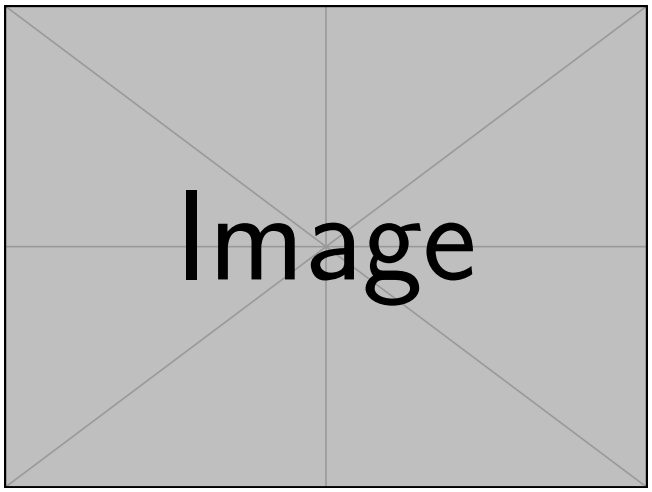

Рис. 1: Пример изображения.

7 Приложение

7.1 Исходный код c++

7.1.1 Последовательная реализация

```
int orientation(Point p, Point q, Point r) {
  int val = (q.y - p.y) * (r.x - q.x) - (q.x - p.x) * (r.y - q.y);
  if (val == 0) return 0; // Collinear
  return (val > 0) ? 1 : 2; // Clockwise or Counterclockwise
}

std::vector<Point> get_convex_hull(const std::vector<Point>& points) {
  int n = points.size();
  if (n < 3) return std::vector<Point>();

  std::vector<Point> hull;

  int l = 0;
  for (int i = 1; i < n; i++) {
    if (points[i].x < points[l].x) {
      l = i;
    }
  }

  int p = l;
  int q;
  do {
    hull.push_back(points[p]);
    q = (p + 1) % n;
    for (int i = 0; i < n; i++) {
      if (orientation(points[p], points[i], points[q]) == 2) {
        q = i;
      }
    }

    p = q;
  } while (p != l);
  return hull;
}
```

7.1.2 OMP

```
point_orientation orientation(Point p, Point q, Point r) {
  int val = (q.y - p.y) * (r.x - q.x) - (q.x - p.x) * (r.y - q.y);
  if (val == 0) return point_orientation::colliniar;
  return (val > 0) ? point_orientation::clockwise
                   : point_orientation::counterclockwsise;
}

double dist2(Point a, Point b) {
  int dx = b.x - a.x;
  int dy = b.y - a.y;
  return dx * dx + dy * dy;
}

std::vector<Point> get_convex_hull_omp(const std::vector<Point>& points) {
  int n = points.size();
  if (n < 3)
    return std::vector<Point>();
  else if (n == 3)
    return std::vector<Point>(points);

  std::set<Point> hull;

  int start = 0;
  for (int i = 1; i < n; i++) {
    if (points[i] < points[start]) {
      start = i;
    }
  }
  int current = start;
  int next;
  std::vector<int> to_push_by_each_thread;
#pragma omp parallel shared(start, next, current, n)
  {
    while (true) {
#pragma omp single
      { next = (current + 1) % n; }
      int private_next = next;
      bool was_used = false;
#pragma omp for
      for (int i = 0; i < n; ++i) {
        if (orientation(points[current], points[private_next], points[i]) ==
            point_orientation::counterclockwsise) {
          private_next = i;
          was_used = true;
        }
      }
#pragma omp critical
      {
        if (was_used) {
```

```
          to_push_by_each_thread.push_back(private_next);
        }
      }
#pragma omp barrier
#pragma omp single
      {
        for (int i = 0; i < to_push_by_each_thread.size(); ++i) {
          if (orientation(points[current], points[next],
                          points[to_push_by_each_thread[i]]) ==
                  point_orientation::counterclockwsise ||
              orientation(points[current], points[next],
                          points[to_push_by_each_thread[i]]) ==
                      point_orientation::colliniar &&
                  dist2(points[current], points[next]) <
                      dist2(points[current],
                            points[to_push_by_each_thread[i]])) {
            next = to_push_by_each_thread[i];
          }
        }
        to_push_by_each_thread.clear();
        for (int i = 0; i < n; i++) {
          if (orientation(points[current], points[next], points[i]) ==
              point_orientation::colliniar) {
            hull.insert(points[i]);
          }
        }
        current = next;
      }
      if (next == start) {
        break;
      }
#pragma omp barrier
    }
  }
  return std::vector<Point>(hull.begin(), hull.end());
}
```

7.1.3 TBB

```
point_orientation orientation(const Point& p, const Point& q, const Point& r)
    {
  int val = (q.y - p.y) * (r.x - q.x) - (q.x - p.x) * (r.y - q.y);
  if (val == 0) return point_orientation::colliniar;
  return (val > 0) ? point_orientation::clockwise
                   : point_orientation::counterclockwsise;
}

double dist2(const Point& a, const Point& b) {
  int dx = b.x - a.x;
  int dy = b.y - a.y;

  return dx * dx + dy * dy;
}

struct MyFunctor {
  int current;
  int next;
  const std::vector<Point>& points;
  MyFunctor(const std::vector<Point>& input, int next, int current)
      : next(next), current(current), points(input) {}
  MyFunctor(const MyFunctor& other, tbb::split)
      : next(other.next), current(other.current), points(other.points) {}
  void operator()(const tbb::blocked_range<int>& r) {
    for (int i = r.begin(); i != r.end(); ++i) {
      if (orientation(points[current], points[next], points[i]) ==
          point_orientation::counterclockwsise) {
        next = i;
      }
    }
  }
  void join(const MyFunctor& other) {
    if (orientation(points[current], points[next], points[other.next]) ==
            point_orientation::counterclockwsise ||
        orientation(points[current], points[next], points[other.next]) ==
                point_orientation::colliniar &&
            dist2(points[current], points[next]) <
                dist2(points[current], points[other.next])) {
      next = other.next;
    }
  }
};

std::vector<Point> get_convex_hull_tbb(const std::vector<Point>& points) {
  int n = points.size();
  if (n < 3)
    return std::vector<Point>();
  else if (n == 3)
```

```
    return std::vector<Point>(points);

  std::set<Point> hull;
  int start = 0;
  for (int i = 1; i < n; i++) {
    if (points[i] < points[start]) {
      start = i;
    }
  }
  int current = start;
  int next;
  while (true) {
    next = (current + 1) % n;
    MyFunctor temp(points, next, current);
    tbb::parallel_reduce(tbb::blocked_range<int>(0, n), temp);
    next = temp.next;
    for (int i = 0; i < n; i++) {
      if (orientation(points[current], points[next], points[i]) ==
          point_orientation::colliniar) {
        hull.insert(points[i]);
      }
    }
    current = next;
    if (next == start) {
      break;
    }
  }
  return std::vector<Point>(hull.begin(), hull.end());
}
```

7.1.4 std::thread

```
point_orientation orientation(const Point& p, const Point& q, const Point& r)
    {
  int val = (q.y - p.y) * (r.x - q.x) - (q.x - p.x) * (r.y - q.y);
  if (val == 0) return point_orientation::colliniar;
  return (val > 0) ? point_orientation::clockwise
                   : point_orientation::counterclockwsise;
}

double dist2(const Point& a, const Point& b) {
  int dx = b.x - a.x;
  int dy = b.y - a.y;
  return dx * dx + dy * dy;
}

void thread_func(int begin, int end, int current,
                 const std::vector<Point>& points, int* per_thread_next) {
  for (int i = begin; i < end; ++i) {
    if (orientation(points[current], points[*per_thread_next], points[i]) ==
        point_orientation::counterclockwsise) {
      *per_thread_next = i;
    }
  }
}

std::vector<Point> get_convex_hull_std(const std::vector<Point>& points,
                                       int thread_num) {
  int n = points.size();
  if (n < 3)
    return std::vector<Point>();
  else if (n == 3)
    return std::vector<Point>(points);

  std::set<Point> hull;
  std::vector<int> per_thread_next(thread_num, -1);
  int start = 0;
  for (int i = 1; i < n; i++) {
    if (points[i] < points[start]) {
      start = i;
    }
  }
  int current = start;
  int next;
  while (true) {
    std::vector<std::thread> threads;
    next = (current + 1) % n;
    std::fill(per_thread_next.begin(), per_thread_next.end(), next);
    for (uint64_t i = 0; i < thread_num; ++i) {
      int operations_per_thread = (n + thread_num - 1) / thread_num;
```

```
      threads.push_back(
          std::thread(thread_func, operations_per_thread * i,
                      std::min(operations_per_thread * (i + 1), uint64_t(n)),
                      current, std::ref(points), &per_thread_next[i]));
    }
    for (uint64_t i = 0; i < thread_num; ++i) {
      threads[i].join();
    }
    for (uint64_t i = 0; i < thread_num; ++i) {
      if (orientation(points[current], points[next],
                      points[per_thread_next[i]]) ==
              point_orientation::counterclockwsise ||
          orientation(points[current], points[next],
                      points[per_thread_next[i]]) ==
                  point_orientation::colliniar &&
              dist2(points[current], points[next]) <
                  dist2(points[current], points[per_thread_next[i]])) {
        next = per_thread_next[i];
      }
    }

    for (int i = 0; i < n; i++) {
      if (orientation(points[current], points[next], points[i]) ==
          point_orientation::colliniar) {
        hull.insert(points[i]);
      }
    }
    current = next;
    if (next == start) {
      break;
    }
  }
  return std::vector<Point>(hull.begin(), hull.end());
}
```

7.2 Тесты последовательной реализации

```
TEST(convex_hull_test, return_empty_vector) {
  std::vector<Point> input;
  input.push_back(Point(1, 0));
  input.push_back(Point(1, 2));
  auto output = get_convex_hull(input);
  EXPECT_EQ(output.size(), 0);
}

TEST(convex_hull_test, hull_of_three_points) {
  std::vector<Point> input;
  input.push_back(Point(1, 0));
  input.push_back(Point(1, 2));
```

```
  input.push_back(Point(1, 3));
  auto output = get_convex_hull(input);
  std::sort(input.begin(), input.end());
  std::sort(output.begin(), output.end());
  EXPECT_TRUE(output[0] == output[0]);
  EXPECT_TRUE(output[1] == output[1]);
  EXPECT_TRUE(output[2] == output[2]);
}

TEST(convex_hull_test, hull_of_five_points) {
  std::vector<Point> input;
  input.push_back(Point(-1, 0));
  input.push_back(Point(0, 1));
  input.push_back(Point(0, -1));
  input.push_back(Point(1, 0));
  input.push_back(Point(0, 0));
  auto output = get_convex_hull(input);
  std::cout << output.size() << std::endl;
  std::sort(output.begin(), output.end());
  EXPECT_EQ(output.size(), 4);
  EXPECT_TRUE(output[0] == Point(-1, 0));
  EXPECT_TRUE(output[1] == Point(0, -1));
  EXPECT_TRUE(output[2] == Point(0, 1));
  EXPECT_TRUE(output[3] == Point(1, 0));
}

TEST(convex_hull_test, collinear_points) {
  std::vector<Point> input;
  input.push_back(Point(0, 0));
  input.push_back(Point(0, 4));
  input.push_back(Point(4, 0));
  input.push_back(Point(1, 1));
  input.push_back(Point(2, 2));
  auto output = get_convex_hull(input);
  std::sort(output.begin(), output.end());
  EXPECT_EQ(output.size(), 3);
  EXPECT_TRUE(output[0] == Point(0, 0));
  EXPECT_TRUE(output[1] == Point(0, 4));
  EXPECT_TRUE(output[2] == Point(4, 0));
}

TEST(convex_hull_test, only_collinear_points) {
  std::vector<Point> input;
  input.push_back(Point(0, 0));
  input.push_back(Point(1, 1));
  input.push_back(Point(2, 2));
  input.push_back(Point(3, 3));
  auto output = get_convex_hull(input);
  std::sort(output.begin(), output.end());
  EXPECT_EQ(output.size(), 4);
  EXPECT_TRUE(output[0] == Point(0, 0));
  EXPECT_TRUE(output[1] == Point(1, 1));
```

```
  EXPECT_TRUE(output[2] == Point(2, 2));
  EXPECT_TRUE(output[3] == Point(3, 3));
}
```

7.3 Тесты параллельных реализаций (отличаются друг от друга функцией get_convex_hull_[название технологии])

```
void fill_vec(std::vector<Point>* vec, uint32_t size) {
  std::set<Point> unique_points;
  std::random_device dev;
  std::mt19937 rng(dev());
  std::uniform_int_distribution<int> dist(0, size << 1);
  while (unique_points.size() != size) {
    unique_points.insert(Point(dist(rng), dist(rng)));
  }
  std::copy(unique_points.begin(), unique_points.end(),
            std::back_inserter(*vec));
  std::shuffle(vec->begin(), vec->end(), rng);
}

void print_vecs(std::vector<Point> a, std::vector<Point> b) {
  std::cout << "First size is " << a.size() << std::endl;
  for (uint64_t i = 0; i < a.size(); ++i) {
    std::cout << a[i].x << " " << a[i].y << std::endl;
  }
  std::cout << "Second size is " << b.size() << std::endl;
  for (uint64_t i = 0; i < b.size(); ++i) {
    std::cout << b[i].x << " " << b[i].y << std::endl;
  }
}

TEST(convex_hull_test, return_empty_vector) {
  std::vector<Point> input;
  auto output = get_convex_hull(input);
  auto output2 = get_convex_hull_std(input, 4);
  EXPECT_EQ(output.size(), 0);
  EXPECT_EQ(output2.size(), 0);
}

TEST(convex_hull_test, hull_of_three) {
  std::vector<Point> input;
  fill_vec(&input, 3);
  auto output = get_convex_hull(input);
  auto output2 = get_convex_hull_std(input, 8);
  bool dumb_lint1 = output.size() == output2.size();
  bool dumb_lint2 = output == output2;
```

```
  bool dumb_lint3 = output.size() == 3;
  EXPECT_EQ(dumb_lint1, true);
  EXPECT_EQ(dumb_lint2, true);
  EXPECT_EQ(dumb_lint3, true);
}

TEST(convex_hull_test, hull_of_ten) {
  std::vector<Point> input;
  fill_vec(&input, 10);
  auto output = get_convex_hull(input);
  auto output2 = get_convex_hull_std(input, 16);
  if (output.size() != output2.size()) {
    print_vecs(output, output2);
    print_vecs(input, input);
  }
  bool dumb_lint1 = output.size() == output2.size();
  bool dumb_lint2 = output == output2;
  EXPECT_EQ(dumb_lint1, true);
  EXPECT_EQ(dumb_lint2, true);
}

TEST(convex_hull_test, hull_of_hundred) {
  std::vector<Point> input;
  fill_vec(&input, 100);
  auto output = get_convex_hull(input);
  auto output2 = get_convex_hull_std(input, 32);
  if (output.size() != output2.size()) {
    print_vecs(output, output2);
  }
  bool dumb_lint1 = output.size() == output2.size();
  bool dumb_lint2 = output == output2;
  EXPECT_EQ(dumb_lint1, true);
  EXPECT_EQ(dumb_lint2, true);
}

TEST(convex_hull_test, hull_of_thousand) {
  std::vector<Point> input;
  fill_vec(&input, 1000);
  auto output = get_convex_hull(input);
  auto output2 = get_convex_hull_std(input, 64);
  bool dumb_lint1 = output.size() == output2.size();
  bool dumb_lint2 = output == output2;
  EXPECT_EQ(dumb_lint1, true);
  EXPECT_EQ(dumb_lint2, true);
}
```