Рандомизированные алгоритмы

Как и предыдущая, эта тема возникла благодаря поездки автора книги в Сириус. В этой главе мы изучим алгоритмы, главной основой которых являются случайные числа. Применять же такие алгоритмы мы можем, когда в задачи требуется найти приближённое значение какой-то величины, или найти максимально хорошую конструкцию для какой-либо задачи. Понятнее это станет на примерах.

Метод Монте-Карло

Пусть нам задана функция $f(x)$ и требуется вычислить её определённый интеграл: $\int\limits_a^b f(x)\,dx$. Если функция f простая, то такой интеграл можно выразить через другие математические функции, но для сложных функций f такой интеграл может и не посчитаться. Поэтому перед информатиками ставится задача численно вычислять такие величины.

Геометрическая интерпретация интеграла — площадь под графиком, поэтому немного переформулируем нашу задачу. Пусть на плоскости есть какая-то фигура сложной формы и нам требуется узнать её площадь. Тогда нам на помощь приходит *«Метод Монте-Карло»*, который может численно решить такую задачу.

Суть самого метода очень проста: ограничим нашу фигуру какой-нибудь другой, площадь которой S мы точно знаем (например, площадь прямоугольника или окружности). Тогда если сложная фигура будет иметь площадь S_0, то оказывается, что будет верно равенство: $\frac{S_0}{S} = \frac{N_0}{N}$, где N — количество случайно взятых точек внутри простой фигуры, а N_0 — сколько из этих точек попали внутрь сложной фигуры. Тогда площадь сложной фигуры можно выразить: $S_0 = S \cdot \frac{N_0}{N}$.

Корректность этого метода основывается на геометрической вероятности и математическом ожидании, поэтому приводить полное доказательство этого метода в книге не хочется. Из важного же стоит сказать, что если мы хотим получить точность оценки площади $\frac{1}{r}$, то нам потребуется взять $N = r^2$, например, если требуется точность 0.01, то достаточно взять 10000 случайных точек (но можно и больше, хуже не будет).

Для примера, почитаем данным методом площадь прямойгольного треугольника с вершинами $(0;0)$, $(0;1)$ и $(1;0)$, взяв за известную фигуру прямоугольник $(0;0)$, $(0;1)$, $(1;1)$, $(1;0)$:

```cpp
#include <iostream>
#include <random>                                   // для mt19937
#include <chrono>                                   // для chrono

using namespace std;
const int n = 1000000, D = 10000;

int main(){
    mt19937 rnd(chrono::steady_clock::now().time_since_epoch().count());
    uniform_int_distribution<> dist(0, D);          // рандомные числа из [0; D]
    int n0 = 0;
    for(int i = 0; i < n; ++i){
        double x = 1. * dist(rnd) / D, y = 1. * dist(rnd) / D; // обе координаты в [0; 1]
        n0 += (x + y < 1);                          // внутри треугольника?
    }
    cout << 1. * n0 / n;
    return 0;
}
```

Можно поэкспериментировать с количеством точек и убедиться в получаемой точности. Но поскольку в алгоритме используются случайные величины, то он может выдавать немного разные

ответы при новых запусках и точность может варьироваться. Поэтому, если отправлять случайные алгоритмы на проверку несколько раз, то они могут набрать разное количество баллов :)

Локальные оптимизации

Пусть нам нужно решить какую-то сложную задачу, в которой есть несколько решений, но с нас требуется одно найти лишь одно любое. Тогда мы могли перебрать все варианты и проверить, являются ли они решениями, но бывает, что перебирать все варианты долго. Поэтому снова приходится прибегать к случайным числам.

Сгенерируем какое-нибудь решение и будем пытаться его оптимизировать. На каждом шаге оптимизации будем пытаться поменять один элемент в решении и если оно от этого улучшиться, то применим это изменение, а иначе ничего делать не будем. Этот метод и называется *«локальные оптимизации»*. Минусами такого метода является вероятность его захождения в тупик, то есть в такую позицию, которую нельзя улучшить. В таких ситуациях стоит начать поиск решения с начала, или же применить какое-то ухудшающее изменение.

Оказывается, что ухудшающие изменения можно совершать не только при захождении в тупик, но и при обычных попытках оптимизации. Но такие изменения следует делать не очень часто на хороших решениях, чтобы случайно их не ухудшить слишком сильно.

Для этого введём T — текущая «температура» и $f(u)$ — функция, которая возвращает, на сколько решение u хорошо (чем меньше, тем лучше, 0 для наилучшего решения). Тогда если для старое решение old, а новое — new, то улучшающие изменения нужно делать всегда, а ухудшающие только с вероятностью $e^{\frac{f(old)-f(new)}{T}}$. А температуру на каждой оптимизации будем менять по правилу $T' = T \cdot k$, где $k \approx 0.99$, но эту константу нужно подбирать. Данный метод называется *«отжигом»* (или методом *«паяльника»*, потому что горячим паяльником тыкают в хорошие решения, от чего им приходится ухудшаться :)) и с его помощью можно решить какое-то количество задач.

Также, кроме отжига, есть и другие улучшения локальных оптимизаций. Одним из них являются *«генетические»* (или *«эволюционные»*) алгоритмы. Их суть в том, что текущее решение мы скопируем несколько раз и для каждой копии отдельно сделаем набор *«мутаций»* (можно как в методе отжига). После этого проведём *«естественный отбор»* и выберем лучшие мутации. Можно выбирать несколько лучших вариантов, можно один, здесь всё уже зависит от желания :)

Другие применения рандома

Арифметическая прогрессия. Пусть дано $2n$ чисел, среди которых n образуют арифметическую прогрессию, которую требуется найти.

Выберем два случайных числа из $2n$, тогда вероятность, что они оба принадлежат арифметической прогрессии, будет 0.25. То есть спустя 4 таких выбора мы (это математическое ожидание, случайно может понадобиться много таких выборов), найдём пару чисел из арифметической прогрессии. Проверка же, что пара чисел находится в арифметической прогрессии, делается через перебор шага прогрессии среди делителей разности выбранных элементов.

«Градиентный спуск». В нейросетях существует такой метод, как градиентный спуск. Суть его в том, что нам даны какие-то точки в многомерном пространстве и нам нужно так подобрать коэффициенты, чтобы заданная функция была максимально близка к этим точкам. Поскольку параметров очень много, то перебирать их все не получаются и используют математические формулы.

Но недостаток этих формул в том, что их достаточно долго считать, если использовать все данные точки. Поэтому используют только часть, которую можно выбирать случайным образом. ведь если часть не очень маленькая, то она должна быть примерно такой же, как и все данные. Такой градиентный спуск называется *«стохастическим»* (в переводе означает *«случайный»*).