Разработка модели

Для того, чтобы не использовать стохастику не будем моделировать агента. Посчитаем вероятности напрямую.

Для этого составим рекурсивный алгоритм:

В начале вероятность появиться в какой-либо ячейке равна:

$$P_{spawn} = \frac{1}{N \cdot M}$$

Теперь представим, что прошло несколько итераций алгоритма. К следущей итерации агент либо умирает и возраждается в любой ячейке, либо делает шаг в какую-то ячейку.

Вероятность обнаружить агента в ячейке (i, j):

$$P_{i,j} = p \cdot P_{spawn} + (1 - p) \cdot t_{i,j}, \text{ где:}$$

p - вероятнось смерти агента,

$t_{i,j}$ - вероятность прихода агента из другой ячейки.

Агент мог прийти в ячейку с одной из соседних, эти события взаимоисключающие, поэтому общая (апост.: агент не умер) вероятность такого набора событий равна сумме вероятностей отдельных:

$$t_{i,j} = \sum_{neighbor} p^{prev}_{neighbor} \cdot u((i, j), neighbor), \text{ где:}$$

$p^{prev}_{neighbor}$ - вероятность на прошлой итерации обнаружить агента в ячейке $neighbor$,

$u((i, j), neighbor)$ - вероятность того, что агент сделает шаг из ячеки $neighbor$ в (i, j).

Последнее происходит когда, либо агент делает в следствии "равновероятного"шага, лиго в следствии выбора ячейки с наибольшей меткой:

$$u((i, j), neighbor) = q \cdot g((i, j), neighbor) + (1 - q) \cdot w((i, j), neighbor), \text{ где:}$$

q - вероятность сделать "равновероятный"шаг,

$g((i, j), neighbor)$ - вероятность сделать шаг в конкрентную ячеку при "равновероятном"шаге (обратно количеству соседей),

$w((i, j), neighbor$ - индикатор: у этой ячейки наибольшая метка среди соседей прошлой позции?

Запишем в матричном виде:

$$\mathfrak{P} = p \cdot P_{spawn} \cdot \mathfrak{I} + (1 - p) \cdot \mathfrak{T}(\mathfrak{P}^{prev}, \mathfrak{U}), \text{ где:}$$

$\mathfrak{I}$ - матрица, все элементы которой единицы.

Заметим, что только $\mathfrak{P}^{prev}$ имеет динамический характер. Остальные матрицы могут быть посчитаны до начала работы "главного цикла"алгоритма.

Реализация модели

Модель реализована на языке C++, библиотеки с рядом готовых решений не использовались.

Сбока программы

Не делал Makefile/CMake, но проекты такого масштаба и не требуют подобного. Для сборки:

```
g++ src/*.cpp -o /build/prob
```

Запуск соответсвенно:

```
./build/prob 2 2 600 0.1 0.1 #пример
```

Представление входных и выходных данных

Программа получает как аргументы командной строки следующие велечины в порядке:

N - количество строк в матрице

M - количество стобцов

$numOfSteps$ - количество итераций

p - вероятность p

q - вероятность q

Так же в папке $assets/$ должен быть расположен файл с названием $matrixR.txt$. В нем стоит указать матрицу меток. Её размер может быть больше, но точно не меньше заявленного. Все элементы положительны.

Выход программы - матрица вероятностей обнаружения агента в конкретной ячейки, стандартный поток вывода.

Общий концепт

В силу казуальности задачи (не требует большой кодовой базы) было решено "на скорую руку"написать удобной представление матриц: наследники std::vector<std::vector<T>
>.

Был написан класс ProbabilityExplorer, в чьем методе и реализован алгоритм.

При конструировании экземпляра данного класса, ему обязательно передаются следующие параметры:

N - количество строк в матрице

M - количество стобцов

R - матрица меток

p - вероятность p

q - вероятность q

Происходит посчет всего, что можно посчитать заранее, а именно: $g((i,j), neighbor)$ и $w((i,j), neighbor)$.

Особенности реализации

Поведение агента при равенстве меток соседей

В таком случае агент выбирает первого соседа в порядке: левый, нижний, правый, верхний.

Хранение информации о соседе с наибольшей меткой

Решенно было пределать хранить информацию для каждой ячейки о том, является ли она соседом с наибольшей меткой для своего соседа слева/ниже/... То есть для каждой ячейки требовалось хранить четыре флага. Наиболее экономная по памяти реализация: битовые маски.

Проомежуточное представление матриц

Проблемой оказалась ориентация в матрице: нам важно знать не находимся ли мы на границе, ведь мы постоянно обращаемся к соседям со всех возможных сторон. Очевидные решения: множетсвенная проверка или отдельная обработка границ, углов и "внутренностей".

Первое приводит к ветвлениям, что может сильно замедлить выполнение программы, второе - к написанию плохого, однообразного кода (self copypaste).

Более удачным оказалось другое решение: "обрамление"матриц незначищими границами, добавление строк и стобцов в начало и конец. Хоть такое решение и приводит к излишнему копированию, однако это можно выполнить всего раз до входа в основной цикл, а после выхода "извлечь"смысловую часть.