Правительство Российской Федерации
Федеральное государственное автономное
образовательное учреждение высшего образования
«Национальный исследовательский университет «Высшая школа экономики»

Факультет компьютерных наук
Образовательная программа бакалавриата 01.03.02 «Прикладная математика и информатика»

ОТЧЕТ
по учебной практике

на факультете компьютерных наук НИУ ВШЭ
(название организации, предприятия)

Выполнил(а) студент(ка)
группы БПМИ____
Копылов О. И.
(инициалы, фамилия)

(подпись)

Руководитель практики

Департамент больших данных и информационного поиска,
международная лаборатория теоретической информатики, ВНС
(подразделение ФКН, должность)

Гурвич Владимир Александрович
(ФИО руководителя практики)

Дата 23.08.2021 7 (семь) (оценка) (подпись)

Москва – 2021

Оглавление

1. Постановка задачи 3
2. Описание решения 4
3. Анализ результатов 7
4. Выводы 8
5. Источники, материалы 8

1. Постановка задачи

В самом начале практики я познакомился с комбинаторными играми: секи, д-секи, одномастка (стандартная или терминальная, обычная или мизерная), дурак-одномастка, игры НИМ (стандартный, Мура, однородный, симметрический, НИМ на гиперграфе). Отмечу, что изначально планировалось затронуть только первые две игры, но поскольку к команде присоединилось много новых людей, то Владимир Александрович существенно расширил список рассматриваемых игр.

Затем мы обсудили идеи, которые могут помочь в продвижении решений этих игр. Например, обратная индукция и предподсчет. Нашей глобальной целью стало написать сайт, на котором пользователь сможет играть в вышеперечисленные игры против компьютера. При этом компьютер будет ходить оптимально (правильно) и, как предложил Владимир Александрович, на каждом своем ходе выводить множество всех оптимальных ходов.

Забегая вперед, скажу, что сайт, к сожалению, не был реализован, хотя многие из команды свою часть работы выполняли ответственно.

После знакомства с играми полезными идеями, мы распределили, кто над чем будет работать – мне досталась одномастка, а именно (как выяснилось позже) **бэкенд игры «терминальная одномастка» в обычной и в мизерной версиях**. Информация про «одномастку» следует ниже.

Игру "одномастка" предложил Эммануил Ласкер в 1929 году. Числа 1,2,..., 2k написаны на карточках и каким-то образом поделены между двумя игроками (сданы). У каждого ровно k карт. Выбираем, кто ходит первым. После этого игрок выбирает, какой картой (из своих карт, естественно) ходить. Кладет эту карту на стол. Затем второй игрок выбирает, какой картой ходить. Кладет на стол её. Сравнивают, чья карта больше. Будем говорить, что игрок, карта которого больше, "берет взятку". Именно он начинает следующий ход. Две карты, которые до этого лежали на столе, перед следующим ходом сбрасываются в биту (игроки эти карты уже не используют).

В "стандартной одномастке" цель - максимизировать число своих взяток. Побеждает тот, кто взял больше всего взяток.

В "терминальной одномастке" (ДЛЯ КОТОРОЙ НАПИСАНА МОЯ ПРОГРАММА) целью является взять ПОСЛЕДНЮЮ взятку. Тот, кто ее взял, - победитель.

Выше описаны условия для обычной версии игры. В мизерной версии победитель определяется наоборот, а именно:

1. В мизерной "одномастке" победитель - тот, кто взял меньше всего взяток.
2. В мизерной "терминальной одномастке" победитель - это тот, кто НЕ взял последнюю взятку.

2. Описание решения

2.1. Предыстория

Блоком игр, относящихся к одномастке, также занимался мой коллега Евгений Краснов. С ним мы сначала обсудили идеи, касающиеся решения стандартной одномастки и возможных улучшений этого решения (альфа-бетта отсечение, предподсчет). Хотели решить «стандартную одномастку», а затем на основе написанного кода, немного переделав его, решить «терминальную одномастку».

Однако через некоторое время в ходе работы я осознал, что такое решение будет далеко не самым эффективным. Обсудив это, мы с Евгением решили, что он продолжит заниматься стандартной одномасткой, а я буду заниматься терминальной. Для «терминальной одномастки» фактически важна только последняя взятка. Именно благодаря этому существует вполне понятный алгоритм, по которому можно обучить компьютер играть против пользователя. Этот алгоритм я опишу ниже, в «используемых методах».

Прежде всего расскажу, что выполняет моя программа. Пользователь играет против компьютера (моей программы). При этом компьютер играет оптимально: определяет множество всех оптимальных ходов для себя, ходит случайно и равновероятно одним из этих оптимальных ходов. Компьютер наперед просчитывает, кто победит при правильной игре обоих игроков.

На каждом ходе выводится ожидаемый победитель (при правильной игре и пользователя, и компьютера). Компьютер всегда играет правильно, а вот пользователь – необязательно.

Следовательно, возможны такие ситуации:

1. Компьютер при изначально проигрышной для себя ситуации допускает возможность ошибки пользователя. Поэтому компьютер учитывает в своих ходах предполагаемую ошибку пользователя и ходит так, чтобы в случае ошибки пользователя победить. Если пользователь играет правильно, то он одержит победу; а если пользователь допустит ошибку, то победит компьютер.

2. Если же у компьютера изначально есть выигрышная стратегия, то он сразу реализует одну из них (как правило, таких выигрышных стратегий несколько). В таком случае пользователь гарантированно проиграет.

Также программа перед каждым ходом компьютера выводит все его оптимальные ходы и, как я уже сказал выше, выбирает случайный ход из множества оптимальных. Естественно, перед началом игры мы определяем, в какую версию терминальной одномастки мы играем: в обычную или в мизерную; затем задаем, какие карты кому принадлежат.

2.2. Используемые методы

По существу, моя программа представляет собой класс Game с функциями *beginning* и *process, attack* и *defence*. При этом последние две функции написаны в двух видах: для обычной и мизерной версий «терминальной одномастки».

Функция *beginning* нужна, чтобы выбрать версию терминальной одномастки (обычную или мизерную), задать принадлежности карт, выбрать, кто ходит первым (компьютер или пользователь).

Функция *attack* соответствует ходу атакующего игрока, то есть игрока, который ходит первым. Перед его ходом на столе нет карт. Если до этого карты и были на столе, то сейчас они уже в бите.

Функция *defence* соответствует ходу защищающегося игрока, то есть игрока, который ходит вторым, после первого. Перед ходом второго игрока на столе лежит карта первого игрока.

Функция *process* требуется для определения того, кто берет взятку и, как следствие, кто ходит первым в следующем раунде. Также эта функция определяет, закончена ли игра и если закончена, то кто является победителем.

Как я уже сказал выше, в моей программе определяется, во-первых, кто выиграет при правильной обоих игроков, а во-вторых, какие ходы компьютера являются оптимальными. Не умаляя общности, опишу эти алгоритмы для обычной (не мизерной) «терминальной одномастки». Для мизерной версии смысл алгоритмов тот же самый, только максимум меняем на минимум, а минимум на максимум. Итак, рассматриваем обычную «терминальную одномастку».

Во-первых, ожидаемый победитель определяется по принадлежности максимальной карты (из оставшихся в игре). Для этого мы храним множества карт каждого из игроков и массив карт, оставшихся в игре. Отмечу здесь же, что определить, принадлежит ли элемент некоторому множеству, в Python можно за O(1). Игрок, которому принадлежит максимальная карта, является ожидаемым победителем, то есть у этого игрока есть выигрышная стратегия.

Во-вторых, рассматривая карты компьютера, мы можем определить оптимальные ходы по следующему алгоритму:

1) Все карты, кроме старшей, представляют собой оптимальные ходы. Это верно, поскольку мы хотим дать компьютеру взять последнюю взятку и, следовательно, должны оставить карту с лучшими свойствами для конца игры.

2) При этом старшую карту мы тоже включаем в оптимальные, если у компьютера осталась только одна карта (очевидно, что других вариантов сделать ход не существует).

3) Пусть значение старшей карты компьютера равно M. Тогда ее мы тоже считаем оптимальным ходом, если карта со значением m := «Наибольшее значение из оставшихся в игре, которое строго меньшее M». Действительно, карты m и M обладают одинаковыми свойствами в рамках нашей игры:

любая карта, которая принадлежит пользователю и больше m, также больше M. Любая карта, которая принадлежит пользователю и больше M, также больше m.
То же самое верно, если в предыдущих двух предложениях все слова «больше» заменить на «меньше». Поэтому в таком случае даже ход старшей картой является оптимальным.

Корректность определения оптимальных ходов обосновывается доказательством алгоритмов выше.

Для того, чтобы случайно выбирать карту из возможных оптимальных ходов, я использую библиотеку random.

В моей программе функции класса Game вызывают одна другую и в нужный момент, когда карты уже закончились, рекурсия тоже подходит к концу. Корректность работы самих функций проверял на своих тестах на случай, если я где-либо допустил опечатку при реализации доказанного выше верного алгоритма. В двух местах нашел и исправил опечатки.

Дополнительно код не ускорял, потому что асимптотическое ускорение далее невозможно. Про это подробнее написано в «Анализе результатов».

2.3. Код

Код бэкенда игры «терминальная одномастка» в обычной и мизерной версиях:

https://github.com/Oleg13Kopylov/terminal-odnomastka/blob/main/Terminal%20odnomastka%20latest%20version.py

3. Анализ результатов

Пусть, как и в пункте «Постановка задачи», k – это число карт у каждого из игроков в самом начале игры.

Программа использует O(k) памяти и работает за время O(k^2).

Действительно, все переменные, массивы, множества, которые я использую в программе, имеют либо константный размер, либо размер, пропорциональный k. Поэтому программа использует O(k) памяти.

Касаемо времени, сначала вызывается функция *beginning*(только один раз!), она имеет сложность O(k). Затем последовательно вызывается O(k) «комбинаций» функций класса Game. «Комбинация» соответствует ходам компьютера и пользователя, определению игрока, который «берет взятку». «Комбинация» состоит из вот такой последовательности функций: *attack*, *defence, process*. Сложность каждой функции не превосходит O(k), при этом в каждой комбинации всегда присутствует функция, которая соответствует ходу компьютера. Сложность такой функции O(k). Следовательно, программа работает за время O(k) * O(k) = O(k^2).

Такой эффективности по памяти и по времени достаточно: можно обработать k около тысячи.

Более того, достигнута наилучшая возможная временная сложность алгоритма (программы) для решения поставленной задачи. Действительно, нам необходимо обработать k ходов компьютера, при этом выводя все оптимальные варианты карт на каждом ходе компьютера. Из алгоритма определения оптимальных ходов (см. страницу 5, ближе к концу, после «Во-вторых») явно следует, что таковых карт либо n, либо n-1, а также что множество оптимальных ходов всегда не пусто. Здесь n – это число оставшихся карт у компьютера. Значит, суммарно необходимо вывести не менее (k-1) + (k-2) + (k - 3) + (k - 4) + ... 1 = k*(k-1)/2 = O(k^2) карт. Поэтому любая программа работает не быстрее, чем за O(k^2) времени. А нам удалось достичь этой сложности!

4. Выводы

Прежде всего, я научился сначала думать, и только после этого реализовывать программу, а не браться за первую идею, которая кажется разумной. Понял, что даже если нечто кажется частным чего-то другого, то не всегда, эффективно реализовав целое, можно достичь столь же эффективной реализации частного.

Узнав про комбинаторные игры подробнее, запрограммировал свою первую игру на Python, тем самым освежив знание этого языка. Еще раз убедился в пользе очевидных, понятных названий для структур данных и функций, особенно когда код получается большим. Научился работать с github.

Получил интересный опыт работы над чем-то длительным: тем, что по времени не ограничивается парой часов, как контесты или олимпиады. Это совершенно другие эмоции.

Достаточно грустно, что сайт так и не написали. Хоть фронтенд и был не моей областью работы в рамках этой учебной практики, я бы не хотел вот так просто отмахнуться от идеи создания сайта и свалить всю вину на фронтэндщиков. Было бы очень здорово, на мой взгляд, довести дело до конца в *полной* мере: создать настоящий сайт с комбинаторными играми. Именно такую цель я ставлю перед собой на летнюю практику следующего года.

5. Источники, материалы

Для знакомства с рассматриваемыми на практике комбинаторными играми использовались следующие ссылки:

1) https://www.sciencedirect.com/science/article/pii/S0012365X1400137X
2) https://math.stackexchange.com/questions/518436/normalizing-a-matrix-with-row-and-column-swapping
3) https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D1%8C%D1%84%D0%B0-%D0%B1%D0%B5%D1%82%D0%B0-%D0%BE%D1%82%D1%81%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5

Для решения конкретно своей задачи, то есть терминальной одномастки, мне Интернет не потребовался: условие задачи рассказал руководитель.
Я разве что освежил знания Python в процессе написания кода, использовав:

1) https://stackoverflow.com/
2) https://www.python.org/

Идею создания класса с функциями, соответствующими действиям в игре, и алгоритмы, необходимые для написания эффективной программы, придумал самостоятельно.