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 БПМИ2010\_\_\_\_\_

|  |
| --- |
| Е. М. Краснов |
| (инициалы, фамилия) |
|  |
| (подпись) |

**Руководитель практики**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Департамент больших данных и информационного поиска / Международная лаборатория теоретической информатики, Ведущий научный сотрудник | | | | | |
| (подразделение ФКН, должность) | | | | | |
| Владимир Александрович Гурвич | | | | | |
| (ФИО руководителя практики) | | | | | |
| Дата |  |  |  |  |  |
|  |  |  | (оценка) |  | (подпись) |

Москва – 2021

Оглавление

[1. Аннотация 3](#_Toc81011311)

[2. Постановка задачи 3](#_Toc81011312)

[3. Описание решения 3](#_Toc81011313)

[4. Анализ результатов 8](#_Toc81011314)

[5. Выводы 10](#_Toc81011315)

### Аннотация

В рамках летней учебной практики, которую я проходил на ФКН НИУ ВШЭ, я в команде с другими студентами работал над проектом «Игра Секи и другие комбинаторные игры». Сначала планировалось, что мы будем работать только над играми Секи и Д-Секи, но нас оказалось слишком много, поэтому наш руководитель, Владимир Александрович, включил в проект ещё игры. В итоге то, чем я реально занимался на практике, по независящим от меня причинам несколько отличается от того, что было заявлено в Индивидуальном Задании.

Я работал над комбинаторной игрой «Бридж-Одномастка». Коротко опишу правила.

Играют 2 игрока. Фиксировано некоторое число k. Числа 1,2,...,2k написаны на карточках и  каким-то образом поделены между двумя игроками (сданы). У каждого по k карт. В свой ход игрок выкладывает в открытую любую из своих карт. Другой игрок смотрит, что положил противник, и аналогично делает ответный ход.

Чья карта старше, того и **взятка**. Сыгранные карты уходят из игры. Следующим ходит тот игрок, кто взял последнюю **взятку**. Задача каждого игрока: взять как можно больше взяток.

Игра кончается, когда кончаются карты у игроков.

### Постановка задачи

Мне нужно было написать программу, которая принимает на вход k (k ≤ 9) и определённую раздачу карт между игроками, а потом играет с пользователем в Бридж-Одномастку. Причём программа, конечно, должна играть по-умному, выбирая из всех возможных ходов самые лучшие. Также пользователь может выбрать, кто делает первый ход: он или компьютер.

Кроме того, от меня требовалось реализовать не только классическую версию Бридж-Одномастки, но и несколько альтернативных режимов игры:

1. Мизерная версия: цель игры не максимизировать, а, наоборот, минимизировать число своих взяток.

2. Режим с весами: каждая взятка в зависимости от того, какая она по счёту в игре, имеет некоторый вес, т. е. приносит взявшему её игроку определённое, заранее

известное кол-во очков (это число может быть как положительным, так и отрицательным). Цель игры: максимизировать (минимизировать, если одновременно с этим режимом выбрана мизерная версия) суммарное число своих очков.

Также необходимо было сделать возможным для пользователя не играть в игру самому, а посмотреть, как программа играет за обоих игроков при заданных параметрах. (Естественно, за обоих игроков программа должна делать наилучшие возможные ходы). Разумеется, ни один из «игроков-ботов» не знает заранее, как пойдёт его соперник.

### Описание решения

#### Особенности реализации

Проект написан на языке Python 3. Программа реализована без интерфейса, запуск осуществляется непосредственно через среду разработки (наложить на мою программу интерфейс было задачей другого участника практики). Ввод всех параметров игры осуществляется через файл, куда пользователь записывает настройки, а потом запускает программу. Такой формат на практике оказался самым удобным, так как можно не вводить каждый раз заново все параметры, а изменить в файле те, которые хочешь поменять. Игроки обозначаются как 0 и 1. Пользователь может выбрать, за кого он играет: за 0 или за 1. Внутри программы игроки тоже обозначаются как 0 и 1. Раздачу карт можно задать 2-мя способами: либо числом k и набором карт на руке у одного из игроков (карты второго игрока после этого определяются однозначно), либо двоичным вектором (строкой нулей и единиц) длинной 2k, где i-тый символ – игрок, у которого карта с числом i. Примечательно, что на самом деле компьютер всегда играет за 0, а человек за 1. Когда пользователь выбирает, что хочет играть за 0, мы просто меняем в двоичном векторе все 0 и 1 местами, и меняем порядок хода.

После того, как программу запускают, она делает **предпросчёт** игры. То есть для каждого возможного хода противника программа находит и запоминает наилучший ответный ход.

Таким образом, мы храним дерево игры, в котором для противника рассматриваем все возможные ходы, а для себя только самые лучшие.

После этого начинается сам процесс игры. Когда ход пользователя, программа предлагает ему ввести, какой картой он ходит. А когда ход компьютера, программа просто «достаёт» из памяти запомненный ею ранее наилучший ход для текущей позиции и выводит его. Также после каждого своего хода программа выводит наилучший результат, которого пользователь может добиться, если в дальнейшем не будет ошибаться. Так, в конце игры человек может оценить, насколько хорошо он сыграл и в каких ходах ошибся.

#### Используемые методы

* + 1. ***Объектно ориентированное программирование***

В своей программе я активно использовал парадигму ООП. Как я уже говорил, в памяти хранится дерево игры. Вершинам этого дерева (т. е. позициям получающимся в игре при определённых ходах) соответствуют объекты классов Position и InterPosition. Класс Position – это ситуация в игре, когда игрок, забравший последнюю взятку или ходящий первым, если это только начало игры, должен пойти. А InterPosition – это когда один из игроков положил карту «на стол», и второй должен ему ответить. В дереве игры Position и InterPosition всегда чередуются. Корень дерева – обязательно Position.

Предпросчёт и сам процесс игры осуществляется объектом класса DeepGreen (назван в честь программы Deep Blue, впервые обыгравшей человека в шахматы). Когда программа играет за обоих игроков, создаётся объект класса CompWithCompGame, который в свою очередь создаёт 2 объекта класса DeepGreen, и они играют друг с другом (каждый из них делает свой предпросчёт).

* + 1. ***Метод обратной индукции***

Одним из столпов моего проекта является метод обратной индукции: если позиция конечная, то найти для неё наилучший ход – задача тривиальная; когда же нам надо найти лучший ход для какой-то неконечной позиции, мы рассматриваем все позиции, которые можно получить из текущей, а для них у нас уже всё посчитано. Из возможных результатов выбираем самый лучший. В программировании такой приём удобнее всего реализовать с помощью рекурсии. В данном случае функция принимает на вход некоторую позицию и сразу вычисляет результат, если позиция является конечной. Иначе, она запускает саму себя для всех позиций, которые можно получить из этой. А потом использует полученные для «дочерних» позиций результаты, чтобы вычислить и запомнить результат для текущей.

Примечание: результатом для позиции называем оптимальный ход из неё, счёт игроков в конце игры, если в дальнейшем оба будут играть оптимально, и некоторую другую дополнительную информацию.

Чтобы ускорить работу программы, я использовал рекурсию с запоминанием: в начале работы функции, программа проверяет, не записан ли уже в памяти результат для этой позиции, и если записан, то делает return.

* + 1. ***Правило Минимакса***

Пожалуй, самой главной идейной частью моей программы является Правило Минимакса.

Расскажу, в чём оно заключается. При построении дерева игры мы ищем за себя самые лучшие ходы, а за противника самые худшие для нас (для него-то они как раз самые лучшие). Так как мы должны не рассчитывать на слабого противника, а быть готовы к наихудшему для нас повороту событий. Посмотрим на примере, почему это так важно. Пусть есть позиция P, сейчас наш ход и мы хотим понять, как нам ходить. Для простоты будем считать, что до конца игры осталось всего 2 хода: 1 наш и 1 противника. Пусть есть 2 возможных наших хода: А и В. После хода А, как бы не пошёл противник, мы получим хороший результат. После хода В, если противник ошибётся, мы получим очень хороший результат, лучше, чем при ходе А. Но если после хода В противник пойдёт правильно, мы получим очень плохой результат. Как же нам ходить? По правилу Минимакса мы ходим А – лучший возможный ход при расчёте на **оптимальную** игру противника. Таким образом, когда мы рассматриваем свои ходы, нам нет смысла продолжать дерево из неоптимальных ходов - всё равно мы так не пойдём. Это позволяет сильно ускорить предпросчёт. Но при рассмотрении ходов противника мы не можем поступить аналогично. Да, мы должны быть готовы к наихудшему для нас ходу противника, но ведь мы не можем быть уверенны, что он пойдёт именно так. Может получиться, что противник пошёл не оптимально и получил позицию, для которой мы не сделали предпросчёт, так как оборвали на ней дерево. То есть получиться, что программа не будет знать, как ей ходить. Такого мы допустить не можем. Поэтому за противника нам придётся продолжать строить дерево для **каждого** его возможного хода (хоть рассчитывать мы будем именно на оптимальный).

* + 1. ***Минимакс на практике***

В коде для реализации Минимакса я использую функцию f. Точнее f – это функция с точки зрения математики, в программе она реализована как словарь. Аргументами f являются позиции (объекты классов Position и InterPosition). f(p) – это число, равное **разности количеств взяток программы и её противника в конце игры** при дальнейшей оптимальной игре обоих. Причём считаются только последующие взятки (те, которые находятся ниже чем p в дереве игры). Также есть функция g (это уже функция и с точки зрения математики и с точки зрения программирования), её аргументы – тоже позиции.

Пусть P – это родительская позиция, а p – дочерняя.

g(p) = f(p) + x, где

х = 1, если при переходе от P к р взятку берём мы,

x = -1, если взятку берёт противник,

х = 0, во всех остальных случаях.

Таким образом, функция g позволяет очень хорошо охарактеризовать ход.

Очевидно, что, чем больше g(p), тем лучше этот ход для программы и тем хуже он для её соперника. И наоборот. Более того, выполняется следующий инвариант:

Договоримся, чтоp – это ход, приводящий к позиции p.

Тогда:

g(p1) > g(p2) <=> ход р1 лучше, чем p2

Когда мы обрабатываем позицию, в которой наш ход, мы стараемся максимизировать значение f. Пусть P – текущая позиция, в которой наш ход; p1, p2, …, pn – возможные дочерние позиции. Тогда f(P) = max(g(p1), g(p2), …, g(pn)).

Когда же мы обрабатываем позицию, в которой ход противника, мы, наоборот, минимизируем значение f. То есть f(P) = min(g(p1), g(p2), …, g(pn)).

При реализации этой идеи на практике я столкнулся с серьёзной проблемой: в питоне ключами для словаря могут быть только неизменяемые объекты. Поэтому использовать в качестве ключей для словаря f объекты моих классов оказалось невозможным. Я решил эту проблему следующим образом: написал специальную хеш-функцию h, которая принимает объект класса Position или InterPosition и возвращает его уникальный хеш неизменяемого типа (на самом деле не совсем уникальный, но некоторые позиции можно считать одинаковыми, подробнее об этом позже). А уже по хешу мы кладём значение в f. Хеш этот представляет из себя кортеж из маски позиции, номера игрока, которого сейчас ход, и номер бита маски, который соответствует карте, которая сейчас выложена «на стол» и «ждёт» ответного хода (если такой нет, то это число равно -1).

* + 1. ***Маска позиции***

Расскажу подробнее о том, что такое маска позиции. Определим её по индукции. Для начальной позиции в игре маска – это число, в двоичном представлении в точности совпадающее с бинарным вектором, введённым пользователем (или сгенерированным программой, если пользователь выбрал другой вариант ввода данных). Когда образуется новый InterPosition, его маска наследуется от родительского Position, а в поле table записывается бит карты, которой ходят. Когда же образуется новый Position, в table записывается -1, берётся копия маски родительского InterPosition, из её двоичного представления удаляются биты карты, записанной в родительском table, и карты, которой ходят в ответ (это соответствует моменту, когда 2 карты уходят из игры), полученное число и является новой маской. И т. д. Например, пусть k = 3 и в начале игры карты розданы следующим образом:

0: 2 4 6

1: 1 3 5

Тогда двоичный вектор игры: 101010

Тогда в начальной позиции Р0: mask = 101010, table = -1 (маски сразу запишем в двоичном представлении)

0 начинает и ходит картой 4. Образуется позиция Ip: mask = 101010, table = 3 (номер бита маски, соответствующего карте 4).

1 отвечает картой 5. Образуется позиция P1: mask = 1010, table = -1 (удалили биты маски, соответствующие картам 4 и 5 – эти карты вышли из игры).

* + 1. ***Позиции, которые можно считать одинаковыми***

Несложно понять, что выполняется следующий инвариант:

h(p1) = h(p2) ⬄ f(p1) = f(p2)

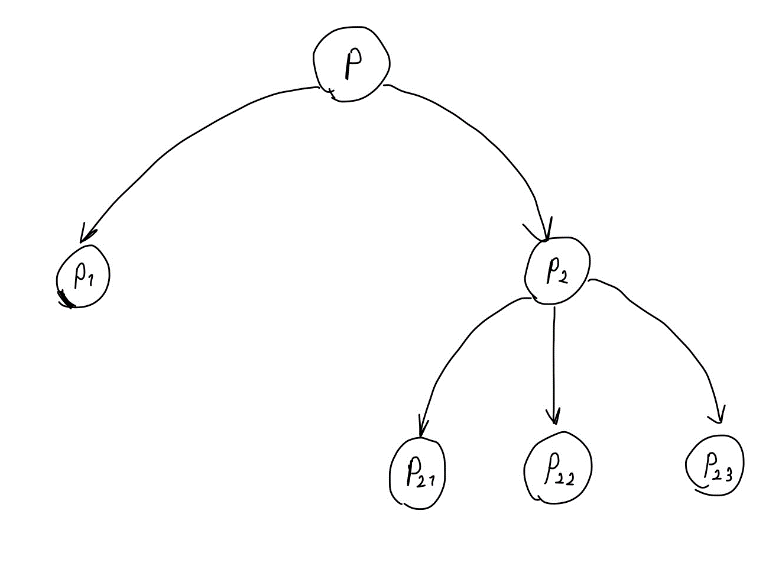
Так как мы используем h только для того, чтобы работать со значениями f, этого нам вполне достаточно и выполнение условия h(p1) = h(p2) ⬄ p1 = p2 вовсе не требуется. Наоборот, даже хорошо, что разные позиции могут иметь одинаковый хеш. Это позволяет сильно оптимизировать программу по времени (увеличится количество случаев, когда мы при входе в функцию, сразу выходим из неё, т. к. значение f для этого ключа уже посчитано).

* + 1. ***Предпросчёт степеней двойки***

При удалении битов из маски программа производит некоторые математические операции в двоичной системе счисления. При этом используются степени двойки. Предпросчёт всех степеней двоек, которые могут понадобиться, методом динамического программирования дал небольшие улучшения по времени.

* + 1. ***Метод ветвей и границ***

Хорошей оптимизацией стало добавление алгоритма «метод ветвей и границ». Коротко расскажу, в чём он заключается. Рассмотрим следующий фрагмент дерева игры:



В позиции p наш ход. Пусть мы закончили обработку дочерней позиции р1 и теперь знаем g(p1) – это ожидаемый результат игры при ходе р1. Начинаем обработку р2. Обрабатываем p21. Видим, что g(p21) + p2.x < g(p1) (p2.x – очко, которое приносит р2, равно либо 1, либо -1, либо 0). Значит, если после нашего хода р2 противник пойдёт р21 мы получим результат гарантированно хуже, чем при нашем ходе р1. Мы должны быть готовы к любому ходу противника, значит, мы точно не будем ходить р2. Всё, ход р2 нам больше не интересен. Значит, поддеревья с корнями в р22 и р23 можно не рассматривать. Поэтому мы делаем выход из функции, обрабатывающей p2, сразу после обработки р21. То есть алгоритм заключается в «отсечении» целых поддеревьев дерева игры, когда мы точно понимаем, что никогда в них не попадём. На практике такие «отсечения» происходят часто, и это позволяет добиться сильного выигрыша по времени.

* + 1. ***Мизерная версия, мизерные изменения***

.Мизерная версия работает абсолютно аналогично обычной. Только цель игры меняется на противоположную и за счёт этого, там, где мы раньше максимизировали, надо минимизировать, и наоборот. На практике это проявляется в том, что константы MIN (абсолютный минимум, начальное значение всех максимумов в моей программе) и MAX (аналогично) меняются местами, и знаки >, < заменяются на противоположные. Чтобы не переписывать один и тот же код 2 раза, что в программировании является плохим тоном, я написал специальную функцию-метод для DeepGreen: less(a, b), которая возвращает a < b в обычной версии и a > b в мизерной. Так, в программе большинство знаков < заменилось на less. Аналогично я ввёл функцию greater(a, b).

* + 1. ***Как добавить веса?***

При игре с весами мы храним массив весов каждой взятки, а функция g определяется следующим образом.

g(p) = f(p) + x, где

х = w[i], если при переходе от P к р взятку берём мы,

x = -w[i], если взятку берёт противник,

х = 0, во всех остальных случаях,

где w[i] – вес текущей взятки. Всё остальное остаётся неизменным.

#### Использованные источники

* Лекции руководителя практики В. А. Гурвича
* <http://algolist.ru/games/alphabeta.php>

### Анализ результатов

В результате практики я выполнил все поставленные задачи. Изначально требовалось, чтобы программа работала при k ≤ 9, но благодаря многочисленным оптимизациям получилось даже перевыполнить этот план. Если k не превосходит 15 (12, если играется режим с весами), то гарантируется, что предпросчёт займёт не более 3 минут.

Но конечно это не предел. Много интересных примеров игры, в которых k > 15, а исследовать их пока нет возможности. Теоретически то программа работает при любых k, но с определённого момента предпросчёт занимает столько времени, что дождаться его окончания нет возможности. Возможно, в дальнейшем можно будет ещё оптимизировать алгоритм.

Ещё из недостатков: режим с весами работает медленнее, чем обычный. Насколько я понял, это из-за того, что в режиме с весами метод ветвей и границ оказывается менее эффективным: меньше случаев, когда можно сделать преждевременный return, не дообработав позицию до конца.

В качестве сильной стороны могу выделить то, что благодаря предпросчёту программа отвечает на ходы пользователя мгновенно.

Также очень удачной оказалась фича с выводом пользователю после каждого хода, какой результат он может получить при дальнейшей оптимальной игре.

Интересно, что программа работает намного быстрее, если в двоичном векторе игры много одинаковых цифр подряд. Дело в том, что, если у игрока на руке несколько подряд идущих карт, то какой из них ходить не имеет значения – это по факту один и тот же ход. Поэтому при построении дерева игры я рассматриваю только один ход из набора подряд идущих карт. Таким образом, при большом количестве подряд идущих карт у каждого из игроков дерево игры получается намного меньше. Соответственно и предпросчёт работает быстрее.

В заключение продемонстрирую, как выглядит работа моей программы.

Примечание: то, что выводит программа;

**то, что вводит пользователь.**

Двоичный вектор: 0 0 0 1 1 1 1 0 0 1 0 1

Карты 0: 4 5 6 7 10 12

Карты 1: 1 2 3 8 9 11

Ход 0: 10 Все оптимальные ходы: 4 5 6 7 10 12 | Взятий нет | Пропускания: 4 5 6 7 10 | Ожидаемое кол-во взяток у 0: 3 | Ожидаемое кол-во взяток у 1: 3

Ход 1: **11**

Ход 1: **9**

Ход 0: 12 Других оптимальных ходов нет | Ожидаемое кол-во взяток у 0: 4 | Ожидаемое кол-во взяток у 1: 2

Ход 0: 4 Все оптимальные ходы: 4 5 6 7 | Взятий нет | Пропусканий нет | Ожидаемое кол-во взяток у 0: 4 | Ожидаемое кол-во взяток у 1: 2

Ход 1: **1**

Ход 0: 5 Все оптимальные ходы: 5 6 7 | Взятий нет | Пропусканий нет | Ожидаемое кол-во взяток у 0: 4 | Ожидаемое кол-во взяток у 1: 2

Ход 1: **2**

Ход 0: 6 Все оптимальные ходы: 6 7 | Взятий нет | Пропусканий нет | Ожидаемое кол-во взяток у 0: 4 | Ожидаемое кол-во взяток у 1: 2

Ход 1: **8**

Ход 1: **3**

Ход 0: 7 Других оптимальных ходов нет | Ожидаемое кол-во взяток у 0: 4 | Ожидаемое кол-во взяток у 1: 2

0 Победил!

### Выводы

Во время практики я научился интересным алгоритмам: правилу минимакса, методу ветвей и границ, познакомился с некоторыми концепциями теории игр и впервые создал собственный программный продукт.

В продолжение практики было бы здорово увеличить максимальное значение k, при котором работает программа. Это можно сделать, например, заранее сделав предпросчёт для всех возможных раздач при всех k от 16 до некоторого выбранного значения (такого, чтобы предпросчёт не занимал больше нескольких суток), а потом записав результаты в базу данных. Тогда в начале игры, если k > 15, мы будем просто доставать дерево игры из соответствующей ячейки базы данных – это будет работать мгновенно.