12. Сим

Постановка задачи

Игра Сим изобретена в 1969 году Густавом Симмонсом, известным американским криптографом.

В игру можно играть на листе бумаги. В начале игры на бумаге по кругу ставится несколько точек, обычно 6.

Затем игроки по очереди ходят. Каждый ход состоит в закрашивании линии, соединяющей две точки, в цвет игрока. В задании будут использованы красный цвет для игрока, который ходит первым, и зелёный для второго. Перекрашивать уже покрашенные линии и пропускать ход нельзя. Проигрывает тот игрок, после хода которого образуется треугольник, со сторонами, проведёнными этим игроком (учитываются только треугольники, вершинами которых являются исходные точки поля; треугольники, образовавшиеся при пересечении линий внутри окружности, не учитываются).

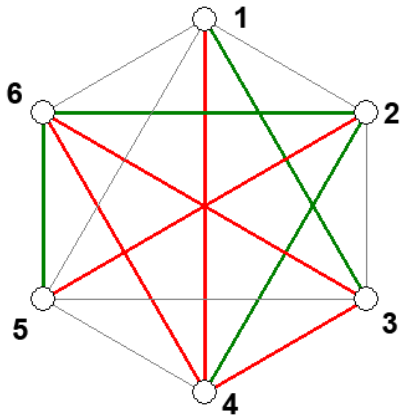

Рис. 12.1 Игра Сим: «красный» игрок проиграл, построив треугольник 3-4-6

Эта игра является результатом исследования математической теории, известной под названием теории Рамсея. В математической постановке задачи точки называют **вершинами** (англ. vertex), а соединяющие их линии – **рёбрами** (англ. edge). Совокупность рёбер и вершин называют **графом** (англ. graph).

С помощью методов теории Рамсея доказано, что в игре Сим с 6 вершинами невозможна ничья, так как любая двухцветная раскраска полного

графа[1] с 6 вершинами обязательно включает треугольник одного цвета. Это свойство не выполняется для другого числа вершин.

Известно, что для этой игры существует выигрышная стратегия, однако пока не удалось сформулировать её в такой форме, которую было бы удобно запоминать человеку.

Познакомиться с математическими свойствами игры можно в работе [1].

Целью задания является разработка компьютерного игрока в игру Сим, выбирающего ходы с помощью метода Монте-Карло.

Предоставляемый код

Для разработки кода следует использовать шаблон sim_template.py, который содержит заголовки функций, которые необходимо разработать, и вызов модуля интерфейса.

Вместе с шаблоном предоставляются также файлы, содержащие код, который следует использовать в решении:

- sim_board.py – реализация класса игрового поля;
- sim_gui.py – графический интерфейс игры;
- sim_compare.py – функции для сравнения эффективности стратегий.

Файл sim_board.py

Файл sim_board.py содержит класс `SimBoard`, который реализует игровое поле игры Сим, функцию `other_player()` и ряд констант.

Константы `PLAYER_RED` и `PLAYER_GREEN` используются как идентификаторы первого и второго игрока, соответственно.

Набор констант, начинающихся на `STATE_` используется для описания состояния игры:

- `STATE_PLAY` – идёт игра, у следующего игрока есть возможность сделать ход;
- `STATE_DRAW` – игра закончена ничьёй;
- `STATE_RED` – игра закончена победой первого (красного) игрока;
- `STATE_GREEN` – игра закончена победой второго (зелёного) игрока.

[1] Полным называют такой граф, в котором существуют рёбра между всеми парами вершин. В игру Сим обычно играют именно на полном графе, хотя можно использовать и не полные графы.

Обратите внимание, что значения двух последних констант соответствует идентификаторам игроков PLAYER_RED и PLAYER_GREEN соответственно. Это позволяет определить, выиграл ли тот или иной игрок, используя сравнение состояния игры с идентификатором игрока.

При использовании этих констант в коде следует ссылаться на них по именам, а не прописывать численные значения. Для того, чтобы имена можно было использовать из других файлов следует сначала импортировать файл sim_board а затем ссылаться на эти константы с префиксом sim_board, как показано в следующем примере.

```
import sim_board      # Импортируем файл.
          …
board = sim_board.SimBoard () # Создаём объект игрового поля
          …
if board.get_state() == sim_board.STATE_DRAW     # Проверка, что игра
                                                 # закончилась в ничью.
```

Функция other_player(player) возвращает «другого игрока»: если на вход подать PLAYER_RED, она вернёт PLAYER_GREEN и наоборот. Эта функция может использоваться для задания очерёдности ходов или узнавания идентификатора оппонента в стратегии. Обращаться к ней из других файлов также следует с префиксом: sim_board.other_player().

Класс SimBoard реализует игровое поле, обеспечивает хранение состояния игры и функции для доступа к нему.

В таблице 12.1 представлено описание доступных для использования методов этого класса.

Таблица 12.1. Методы класса SimBoard

Метод и параметры	Описание
__init__(size) size – размер игры (число точек).	Создаёт поле для игры с заданным числом точек.
__str__	Возвращает строку с описанием состояния игры.
copy()	Возвращает независимую копию поля.
get_size()	Возвращает размер игры.
get_state()	Возвращает состояние игры (значение одной из констант STATE_...).

`get_current_player()`	Возвращает идентификатор игрока, который должен сделать следующий ход.
`get_all_edges()`	Возвращает кортеж (тип `tuple`) всех рёбер. Этот кортеж включает как занятые игроками так и свободные рёбра, и всегда одинаков для игры одинакового размера.
`get_free_edges()`	Возвращает множество (тип `set`) свободных рёбер. Это множество следует использовать в стратегии при выборе следующего хода.
`edge_color(edge)` `edge` – ребро (кортеж).	Возвращает идентификатор игрока, которому принадлежит ребро `edge`. Если ребро свободно, возвращается `None`.
`add_edge(edge)` `edge` – ребро (кортеж).	Делает ход: окрашивает ребро `edge` в цвет игрока, который должен ходить. Возвращает `True` если ребро окрашено, и `False` если это нельзя сделать, например ребро уже занято. Распечатывает отладочное сообщение, если ребро задано некорректно. Может приводить к изменению состояния игры.

Вместе с этими методами в классе `BoardState` есть также метод `check_winner()`, который используется методом `add_edge()` для определения, не завершилась ли игра. Вызывать этот метод не следует.

Для представления рёбер класс `BoardState` использует кортежи (тип `tuple`) из двух значений – номеров вершин, которые соединяет это ребро. Вершины нумеруются с 0 до номера, на 1 меньшего размера игры. Обратите внимание, что одно и то же ребро может быть описано двумя разными кортежами, в которых вершины поменяны местами (например `(0, 1)` и `(1, 0)`). Методы класса `BoardState` всегда возвращают кортеж, в котором номер первой вершины меньше номера второй. Метод `add_edge()` корректно обработает ребро, указанное в любом порядке. Ошибкой является указание ребра, в котором оба номера вершины совпадают (например, `(4, 4)`) или если хотя бы один из номеров не является доступным для данной игры номером вершины.

Файл sim_gui.py

В файле sim_gui.py реализован графический интерфейс игры. Этот класс содержит функцию `run_gui()`, класс `SimGUI` и ряд констант, определяющих внешний вид игры.

Функция `run_gui()` запускает интерфейс. Её параметрами являются `game_size` – размер игры и `machine_player` – функция стратегии. Для запуска игры следует подключить файл sim_gui с помощью команды `import` и вызвать функцию `run_gui()`.

```
import sim_gui
    …
sim_gui.run_gui(6, mc_player)
```

Класс `SimGIU` реализует сам интерфейс. Вызывать методы этого класса в ходе решения не потребуется. С методами этого класса можно познакомиться по комментариям в самом файле.

После запуска интерфейса пользователь может сделать ход последовательно щёлкнув мышью по вершинам, которые он хочет соединить. Выделенная первым щелчком пользователя вершина подсвечивается цветом игрока. Если пользователь передумал, он может снять выделение повторным щелчком на той же вершине.

После щелчка пользователя по второй вершине игра добавляет соответствующее ребро и отображает ответный ход компьютерного игрока, определённый функцией стратегии, которая была передана как параметр функции `run_gui()`.

Файл sim_compare.py

Чтобы оценить эффективность разных стратегий можно заставить их играть друг против друга. Файл sim_compare.py содержит функции, которые проводят такие испытания и распечатывают их результаты.

Функция `sim_compare(player1, player2, game_size)` проводит сравнение стратегий `player1` и `player2` на играх размером `game_size`. Она проводит 100 игр (определяется переменной `COMPARE_TRIALS`), в которых за красных играет стратегия `player1`, а за зелёных – `player2`. После проведения игр печатается, сколько раз первая стратегия выиграла, свела игру в ничью или проиграла.

Обратите внимание, что в играх, подобных Сим, важно, какой игрок делает первый ход, поэтому эффективность игры стратегии «за красных» и «за зелёных» может различаться.

В файле sim_compare.py также содержится функция `game()`. Эта функция проводит одну игру-испытание для функции `sim_compare()`. Вызывать её саму в ходе решения задания не потребуется.

Для вызова функции `sim_compare()` подключите файл sim_compare.py с помощью `import` и используйте префикс `sim_compare.`.

Шаблон sim_template.py

Файл sim_template.py содержит каркас той части приложения, которую следует разработать в ходе решения задачи.

Эта часть включает в себя функцию mc_player(), в которой должна быть реализована стратегия компьютерного игрока на основе метода Монте-Карло, и три вспомогательные функции: `mc_trial()`, `mc_update_scores()` и `get_next_move()`. Работа этих функций будет описана далее.

Этот файл также содержит функцию random_player(), реализующую случайную стратегию. Эту функцию можно использовать для понимания того, как стратегия взаимодействует с остальной частью игры и для сравнения эффективности.

Параметрами функций-стратегий, в том числе функции `random_player()` являются `board` – игровое поле, представленное классом `SimBoard`, и `player` – идентификатор игрока, ход которого следует оценить.

Случайная стратегия получает от игрового поля информацию о доступных для хода дугах и случайно выбирает одну из них. Выбранная стратегией дуга должна быть возвращена из функции в виде кортежа.

В конце шаблона находится вызов графического интерфейса со случайной стратегией. После реализации стратегии Монте-Карло следует подставить её в этот вызов вместо случайной стратегии.

Стратегия для игры Сим

Стратегия должна использовать метод Монте-Карло для выбора хорошего хода в игре, начиная с текущей позиции. Общая идея метода состоит в том, чтобы смоделировать большое число игр со случайными ходами, начиная с текущей позиции, и затем использовать результаты этих игр, чтобы выбрать наиболее удачный ход.

Рассмотрим одну из смоделированных игр. Если в ней «наш» игрок выиграл, то можно сделать вывод, что рёбра, которые он в ней использовал, являются благоприятными для нас, а те которые использовал противник – не благоприятными. Напротив, если игра была проиграна – следует избегать использованных в ней рёбер и предпочесть рёбра противника (чтобы не дать ему их использовать).

В целом получается, что всегда целесообразно выбирать для своего хода те рёбра, которые выбирал выигравший игрок, и не выбирать рёбра проигравшего.

Не забудьте, что окончательный вывод надо сделать не по результатам одной игры, а по результатам множества смоделированных игр. Для этого каждому ребру по результатам каждой игры будут начисляться очки, увеличивающие или уменьшающие степень предпочтительности этого ребра для следующего хода. Полученные в результате накопления на большом числе игр очки и следует использовать для выбора следующего хода.

Таким образом, стратегия выбора хода методом Монте-Карло реализуется следующим образом.

1. Начинаем с текущей позиции.
2. Повторять большое число раз:
    a. смоделировать игру со случайным выбором ходов начиная с текущей позиции и до завершения игры;
    b. скорректировать очки рёбер на основе результата игры.
3. Случайно выбрать ребро с максимальными очками из числа доступных для хода[2].

Наиболее сложным среди этих действий является пункт 2b – корректировка очков. Рассмотрим, как оценить очки для одной игры.

Стратегия должна вычислить очки для каждого ребра. Начисляемые по результатам игры очки зависит от того, кто выиграл рассматриваемую игру.

[2] Одинаковое максимальное значение очков могут иметь сразу несколько рёбер, доступных для следующего хода. В этом случае можно выбрать любое из них – стратегия не может сказать, какое из них является лучшим. Возможно, все они действительно являются одинаково хорошими.

Если игра закончилась в ничью[3], то очки не начисляются, так как эта игра не может помочь определить выигрышную стратегию.

Если игра закончилась выигрышем «нашего игрока», то все «наши» рёбра должны получить положительную оценку (равную значению `MC_WIN` в шаблоне), а рёбра оппонента – отрицательную (значение `MC_LOOSE`).

Напротив, в случае проигрыша следует оценить «свои» рёбра значением `MC_LOOSE`, а рёбра оппонента – значением `MC_WIN`.

Не забудьте, что оценка должна вычисляться по результатам всех игр, поэтому полученные за каждую игру очки следует складывать с очками этого ребра, накопленными на данный момент. Начать следует с оценки `0` для всех рёбер.

Рекомендованный порядок выполнения задания

В ходе работы над заданием следует реализовать 4 функции.

Функция `mc_trial()`

Эта функция должна сгенерировать случайную игру начиная из заданного состояния. Единственным параметром функции является `board` – состояние игрового поля в виде класса `SimBoard`.

Реализация этой функции проста – в цикле получайте список свободных рёбер, выбирайте из них случайное, и делайте следующий ход. Не потребуется даже отслеживать очерёдность ходов, так как это автоматически реализуется классом `SimBoard`. Игру следует продолжать до тех пор, пока состояние игры не изменится с `STATE_PLAY` на какое-нибудь другое.

Для выбора случайного ребра можно использовать функцию `random.choice()`. Учтите, что эта функция не может работать с множествами (тип `set`). Чтобы её использовать, преобразуйте множество свободных рёбер в список (тип `list`).

Функция `mc_trial()` не возвращает никакого значения. В результате её действий изменяется объект `board`, переданный ей как аргумент.

Функция `mc_update_scores()`

Параметрами функции `mc_update_scores()` являются:

[3] Это возможно для размеров игры не равных 6.

- `all_edges` – кортеж (тип `tuple`), содержащий все возможные рёбра в игре;
- `scores` – список (тип `list`) текущих очков для каждого ребра;
- `board` – игровое поле на момент окончания игры (объект класса `SimBoard`);
- `player` – идентификатор игрока, с позиции которого оценивается результат игры.

Стратегия хранит информацию об очках каждого ребра в двух структурах данных: кортеж all_edges хранит сами рёбра, а scores – очки за каждое из них соответственно. Например, если `all_edges = ((0, 1), (0, 2), …)` а `scores = [5, -3, …]` то это значит, что ребро `(0, 1)` имеет оценку $+5$, а ребро `(0, 2)` – оценку -3.

Функция `mc_update_scores()` получает на вход текущие значения очков и должна обновить их по результатам игры, представленной полем `board`.

Во-первых, проверьте, не закончилась ли игра ничьей. В этом случае ничего делать не надо.

Если игра окончилась победой одного из игроков, следует скорректировать очки всех использованных в этой игре рёбер. Правила оценки описаны в разделе «Стратегия для игры Сим». Следует оценить игру, ориентируясь на то, что «наш» игрок, этот тот игрок, идентификатор которого передан в параметре `player`.

Функция `mc_update_scores()` не должна возвращать никаких значений. Следует обновить очки прямо в параметре `scores`.

Функция `get_next_move()`

Функция get_next_move() должна выбрать следующий ход, ориентируясь на вычисленную оценку. Параметрами этой функции являются:

- `all_edges` – кортеж (тип `tuple`), содержащий все возможные рёбра в игре;
- `scores` – окончательный список (тип `list`) очков для каждого ребра;

- `board` – игровое поле на текущий момент (объект класса `SimBoard`).

В этой функции следует выбрать для хода одно из рёбер, имеющих максимальную оценку.

Начните с получения списка доступных для хода рёбер.

Выберите среди них рёбра с максимальной оценкой. Таких рёбер может быть много.

Случайно выберите и верните одно из этих рёбер.

Функция `get_next_move()` должна вернуть выбранное ей ребро, в виде кортежа (тип `tuple`) из двух элементов – начальная и конечная вершина.

Функция `mc_player()`

Эта функция использует три предыдущие для того, чтобы сформировать стратегию. Параметрами функции являются:

- `board` – игровое поле на текущий момент (объект класса `SimBoard`);
- `player` – идентификатор игрока, за которого нужно выбрать ход.

В этой функции должна быть реализована стратегия вычисления рационального хода методом Монте-Карло.

Сначала получите у игрового поля список всех рёбер. Сформируйте список с начальными значениями очков за каждое ребро (начальное значение должно быть 0).

Реализуйте цикл перебора случайных игр. Число итераций цикла должно задаваться значением `MC_TRIALS`, определённым в начале шаблона.

В цикле генерируйте продолжение текущей игры, используя функцию `mc_trial()` и оценивайте её результат, используя `mc_update_scores()`. Не забудьте, что функция `mc_trial()` модифицирует переданное ей игровое поле, поэтому не передавайте ей непосредственно текущее поле (оно ещё потребуется в своём исходном виде), а сделайте копию с помощью соответствующего метода.

После цикла выберите оптимальный ход с помощью функции `get_next_move()` на основе вычисленных в цикле очков.

Функция `mc_player()` должна вернуть выбранное для следующего хода ребро в виде кортежа (тип `tuple`) из двух элементов – начальная и конечная вершина.

Оценка стратегии

Реализовав стратегию, проверьте её на себе. Для этого, замените передачу случайной стратегии в интерфейс игры в конце шаблона на стратегию Монте-Карло.

Оцените эффективность стратегии в игре со случайной стратегией. Для этого раскомментируйте обращения к функции `sim_compare()`, расположенные перед вызовом интерфейса. Сообщения о результатах сравнения будут напечатаны в консоли перед запуском игры. Корректно реализованная стратегия не должна оставить случайной почти никаких шансов на победу.

Дополнительное задание

Представленная в шаблоне функция-стратегия random_player() делает действительно случайный выбор. В том числе, она может выбрать ребро, которое немедленно приведёт к проигрышу, даже если у неё есть доступные альтернативы.

Такое поведение, конечно, нельзя назвать разумным. Реализуйте на базе существующей случайной стратегии усовершенствованный вариант, который будет избегать делать проигрышные ходы, если это возможно.

Учтите, что в конце игры вполне возможна ситуация, когда у стратегии не останется возможности сделать ход, который не приведет к немедленному проигрышу. В такой ситуации стратегия, тем не менее, должна сделать какой-то ход.

Сравните эффективность этой стратегии с чисто случайной стратегией и методом Монте-Карло. Она должна выигрывать у первой и проигрывать второй.

Оценка задания.

Реализация `mc_trial()` - 20 %.

Реализация `mc_update_scores()` - 30 %.

Реализация `get_next_move()` - 25 %.

Выбор фиксированной альтернативы (например, первого или последнего) среди рёбер с максимальной оценкой - -10%.

Реализация `mc_player()` - 25 %.

Дополнительные баллы

Реализация улучшенной случайной стратегии - 15 %.

Срок сдачи задания.

Для подгруппы, занимающейся по четвергам: 27 ноября 2014;

для подгруппы, занимающейся по вторникам: 2 декабря 2014.

В случае сдачи задания с опозданием, полученные за него баллы будут уменьшены:

при задержке на 1 неделю: баллы умножаются на 0.9;

при задержке на 2 недели: баллы умножаются на 0.75;

при задержке на 3 и более недель: баллы умножаются на 0.65.