Проект Hard or Easy

*Примечание: Если вы хотите приступить к быстрой реализации, перейдите на страницу 5. На ней приведена краткая инструкция по реализации библиотеки.*

Проект направлен на анализ заданного уровня при условии, что игрок действует по описанной тактике. Проект может быть полезен для решения следующих задач:

1. Выявление наилучшей тактики игрока (например, в карточных играх).
2. Оценка сложности уровня (насколько возможно пройти ваш собственный уровень).
3. Автоматическая генерация уровней.

Ваша игра должна соответствовать следующим условиям.

1. Дискретность. В игровой логике не должно быть непрерывных величин для подпространства, состоящего из всех возможных состояний игры. Любую непрерывную величину следует дискретизировать. В первую очередь это касается времени игры. К примеру, пошаговые игры имеют номер хода игроков вместо непрерывного времени. Непрерывное время можно легко дискретизировать в том случае, если разница между любыми моментами времени заранее известна. Но в играх реального времени в зависимости от мощности вашего компьютера и других факторов следующий момент времени может быть разным. В таких играх необходимо вводить отдельный параметр – время, связанный с номером итерации. Состояний игры будет чрезмерно много в таком случае. Не рекомендуется анализировать программой такие игры.
2. Конечность. Любая величина должна быть из конечного множества. Если существует величина счетного множества, то и состояний в общей сложности будет бесконечное количество. 1 и 2 пункты можно описать в следующем виде: количество всех возможных состояний игры не должно быть бесконечным.
3. Количество возможных действий, которые совершает игрок, должно быть конечным числом. Обычно это следует из конечного множества состояний.
4. Игра не должна быть бесконечной. Если возможны ситуации, когда игра зацикливается, то анализ не будет совершен. Необходимо исключить случаи зацикливаний и другие ситуации, когда игра может никогда не закончиться.

В папке общего проекта вы можете обнаружить 3 директории:

1. Approbator – программа, симулирующая вашу игру.
2. DLL – библиотека, которую необходимо подключить.
3. HardOrEasy – программа, проводящая анализ.

Для начала вам нужно создать проект «Библиотека классов» в Visual Studio или подобной среде разработки и подключить библиотеку ProbAstimator.dll. Далее необходимо реализовать по порядку некоторые абстрактные классы и интерфейсы этой библиотеки.

В первую очередь нужно реализовать класс **WorldState**. Это состояние игры. В этот класс не включаются методы логики игры. Здесь описывается модель игры. И некоторые методы для расчета внутренних параметров.

В этом классе необходимо реализовать следующие методы:

1. WorldState Clone() – Клонирование состояния со всеми его параметрами.
2. bool Equals(WorldState state) – Проверка совпадения состояния с другим.
3. int CalcUserStateType() – Определение типа состояния (-1 – обычное состояние, 0, 1, … состояния, которые привели к завершению игры). Все состояния описываются в статическом поле WorldState.TypeNames. Данное поле представляет собой массив названий всех состояний. По умолчанию создан массив { “Выигрыш”, “Проигрыш” }. Метод CalcUserStateType не должен возвращать значения, которые больше или равны количеству состояний. Если вы возвращаете -1, значит игра продолжается.

Затем необходимо реализовать класс **StateLoader**. Класс должен иметь:

1. Конструктор по умолчанию.
2. Реализацию метода WorldState LoadStartState(). Возвращение состояния игры.

Данный класс отвечает за загрузку состояния игры. Если состояние игры всегда одинаковое, то StateLoader просто возвращает состояние по умолчанию.

Можно вместо класса **StateLoader** реализовать класс **FileLoader**. В этом классе реализуется другой метод:

WorldState ObtainFromLines(string[] lines).

Данный метод преобразовывает набор строк, считанных из текстового файла, в состояние игры. Если вы реализуете класс FileLoader, программа предложит выбрать текстовый файл.

Имеется также статический класс TextFileLoadTools, имеющий методы преобразования строк в объекты и чтения строк из файла.

На последнем шаге, вообще говоря, требуется реализовать интерфейс **IStateCalculator**. Класс должен иметь:

1. Конструктор по умолчанию.
2. Реализацию метода List<KeyValuePair<WorldState, double>> CalcStateProbPairs(WorldState state, int t).

Этот класс универсален, он описывает все возможные состояния игры, которые возникают из заданного состояния, при этом указывая вероятность наступления того или иного состояния. Метод возвращает список пар <Состояние, вероятность>. Но класс небезопасен и реализуется непросто. Если вы решили его реализовать, следует учитывать следующие 2 момента:

1. В списке пар <Состояние, вероятность> сумма всех вероятностей должна быть равна 1. Если это условие не выполняется, то результаты расчетов будут неверными.
2. Каждое новое состояние списка должно представлять собой новый экземпляр класса. Не допускается просто возвращать ссылку на аргумент функции state. Это главная причина, почему класс небезопасен.

В библиотеке имеются 3 готовые реализации интерфейса IStateCalculator.

1. **PlayerStateCalculator<W>** - в игре участвует один игрок, а игровая система детерминировано отвечает на его команды.
2. **SeveralPlayersStateCalculator<W>** - в игре участвует несколько игроков.
3. **PlayerRandomStateCalculator<W>** - игровая логика случайная, и игру играет один игрок.

Классы принимают дополнительные объекты в конструкторе. Вам достаточно просто создать свой класс, наследовать от PlayerStateCalculator<W>, SeveralPlayersStateCalculator<W> или PlayerRandomStateCalculator<W> и передать в конструктор ваши собственные классы, описывающие стратегии игроков и ответчика системы, если он необходим.

Хочется отметить, что все 3 класса шаблонные. В качестве W указывается ваш реализованный класс состояния игры. Это сделано для удобства реализации методов классов.

Для окончательного описания класса StateCalculator потребуется реализовать ещё несколько классов. Чтобы не создавать путаницу в проекте, создайте отдельные пакеты, в которых будут содержаться реализации следующих интерфейсов, если будут необходимы:

1. **IPlayerCommand<W>**
2. **IStrategy<W>**
3. **IStateProcesser<W> (нужен только для PlayerStateCalculator<W>)**
4. **IRandomStateProcesser<W> (нужен только для PlayerRandomStateCalculator<W>)**

В отдельном пакете создайте все команды игроков, которые возможны в вашей игре, то есть реализации интерфейса **IPlayerCommand<W>**. В качестве W аналогично указываете ваш класс с состоянием игры. Команда – это указание игрока на то, что он хочет произвести определенное действие. В интерфейсе команды необходимо реализовать один простой метод:

void Process(ref W state).

Здесь просто в указанном состоянии state меняются какие-либо параметры. Создавать новую копию state не требуется. Удобство заключается в том, что вы напрямую работаете с вашим классом и не пользуетесь приведением типов к классу вашего состояния.

Отдельный пакет будет содержать все стратегии игроков, то есть реализации класса **IStrategy<W>**. Однако, если вы выбрали PlayerStateCalculator<W>, то вы в этом же пакете можете ещё реализовать интерфейс **IStateProcesser<W>**.

**IStateProcesser<W>** имеет похожий метод, как и у команды:

void Process(ref W state, int iteration).

В данном методе только добавился аргумент с номером итерации. Интерфейс отвечает за изменение состояния на данной итерации в ответ на действие игрока.

Пожалуй, самым сложным является реализация интерфейса **IStrategy<W>**, поскольку здесь необходимо проанализировать текущее состояние и указать, как может действовать игрок в такой ситуации. Интерфейс имеет один метод:

CommandProbPair<W>[] CalcCommands(W state, int localIteration).

Метод для заданного состояния и номера итерации возвращает массив пар <команда, вероятность>. Для создания такого массива нужно:

1. из всех возможных команд исключить те, которые не соответствуют правилам игры;
2. из оставшихся команд исключить те, которые не являются осмысленными;
3. указать, с какой вероятностью какую команду из оставшихся игрок предпочтет.

Игрок может по невнимательности совершить ошибку, тогда для такой команды указываем не очень большую вероятность. Игрок может плохо спрогнозировать дальнейшие действия, тогда тоже возникает вероятность. Если для игрока не имеет значения, какое действие совершить, тогда вероятности одинаковы.

Следует отдельно сказать про класс **CommandProbPair<W>**. Создать этот класс можно через конструктор по умолчанию и установку свойств команды и вероятности. Однако есть помощники – статистические методы создания набора пар:

1. CommandProbPair<W>[] GenerateCommand(IPlayerCommand<W> command) – сгенерировать массив, состоящий из одной команды с вероятностью выбора 1.
2. CommandProbPair<W>[] GenerateUniformCommands(IPlayerCommand<W>[] commands) – сгенерировать массив, состоящий из команд с равными вероятностями.

**IRandomStateProcesser<W>** представляет собой обработчик игровой логики, когда игра должна сгенерировать новые состояния. Данный интерфейс необходимо реализовать, если вы решили использовать класс **PlayerRandomStateCalculator<W>** в качестве игры. В таком случае игра представляет собой последовательность из двух шагов на каждой итерации: на первом шаге игра генерирует случайное состояние, на втором шаге ходит игрок.

Вообще говоря **IRandomStateProcesser<W>** не безопасный интерфейс, поскольку никак не контролирует копирование состояния (ситуация та же, что и для **IStateCalculator**). Однако игровая логика, как правило, генерирует новые игровые состояния, в которых сохраняется очень мало параметров, а описание того, как игра подбрасывает случайные события, обычно довольно простое. Поэтому учесть небезопасность данного интерфейса существенно проще, чем IStateCalculator.

Интерфейс содержит один метод:

List<KeyValuePair<WorldState, double>> CalcStateProbPairs(W state, int t).

Данный метод возвращает список пар <Состояние, вероятность>. Как и во всех остальных случаях, сумма всех вероятностей должна быть равна 1. К сожалению, для создания набора пар не предусмотрены методы генерации равновероятных состояний или одного состояния, как это сделано для CommandProbPair<W>. Здесь приходится создавать список вручную.

Если вы планируете произвести симуляцию разработанной игры для проверки корректности, то вам нужно реализовать ещё один интерфейс **IDrawer**. Данный интерфейс содержит один простой метод

void Draw(WorldState state, int iteration).

Метод просто выводит в консоль ваше состояние. Вы реализуете данный метод при помощи методов класса Console. Симулятор будет выводит каждую итерацию игры при помощи данного метода.

После реализации всех описанных классов можете собирать проект и использовать его для анализа.

Приведу краткую инструкцию, если предыдущий текст показался сложным.

1. Создать проект «Библиотека классов» и подключить библиотеку ProbAstimator.dll.
2. Реализовать класс WorldState.
3. Реализовать класс StateLoader.
   1. Или реализовать класс FileLoader.
4. Реализовать интерфейс IStateCalculator.
   1. Или реализовать класс PlayerStateCalculator<W>.
      1. Создать пакет «Commands» для команд.
      2. Реализовать в пакете «Commands» все возможные команды на основе реализации интерфейса IPlayerCommand<W>.
      3. Создать пакет «Strategy» для стратегии игрока.
      4. Реализовать в пакете «Strategy» интерфейс IStrategy<W>.
      5. При необходимости реализовать в пакете «Strategy» интерфейс IStateProcesser<W>.
      6. Указать в классе-наследнике от PlayerStateCalculator<W> реализованную стратегию и ответчика.
   2. Или реализовать класс SeveralPlayersStateCalculator<W>.
      1. Создать пакет «Commands» для команд.
      2. Реализовать в пакете «Commands» все возможные команды на основе реализации интерфейса IPlayerCommand<W>.
      3. Создать пакет «Strategies» для стратегии игроков.
      4. Реализовать в пакете «Strategy» все стратегии игроков IStrategy<W>.
      5. Указать в классе-наследнике от SeveralPlayersStateCalculator<W> реализованные стратегии игроков.
   3. Или реализовать класс PlayerRandomStateCalculator<W>.
      1. Создать пакет «Commands» для команд.
      2. Реализовать в пакете «Commands» все возможные команды на основе реализации интерфейса IPlayerCommand<W>.
      3. Создать пакет «Strategies» для стратегии игроков.
      4. Реализовать в пакете «Commands» интерфейс IRandomStateProcesser<W> - обработчик логики.
      5. Реализовать в пакете «Strategy» интерфейс IStrategy<W> - стратегия игрока.
      6. Указать в классе-наследнике от PlayerRandomStateCalculator<W> реализованный обработчик логики и стратегию игрока.
5. Если планируется симуляция игры, то следует реализовать интерфейс IDrawer.

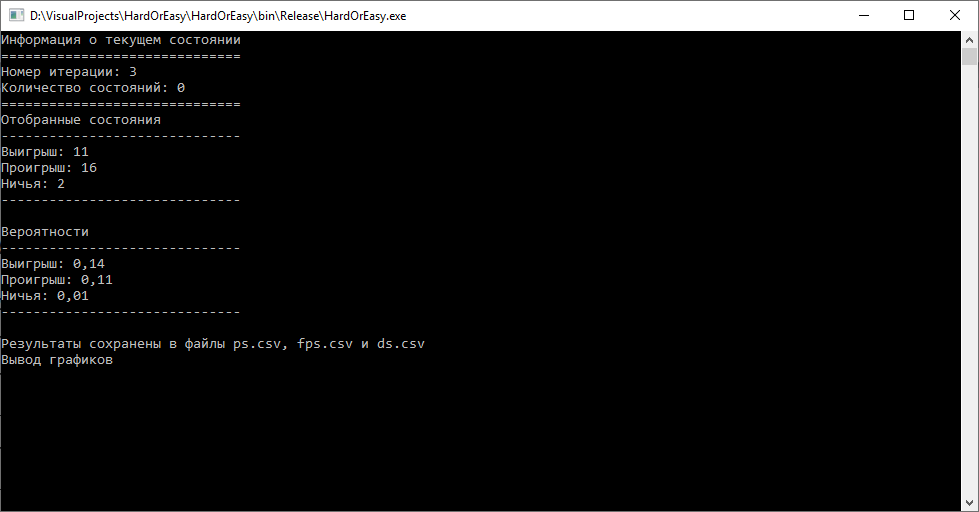
После завершения сборки вам нужно скопировать вашу библиотеку в папку с программой (обе программы Approbator и HardOrEasy работают по одному принципу) и в файле settings.txt изменить имя на имя вашей библиотеки, чтобы программа понимала, какой файл сборки загружать. Далее запускаете программу и делаете все по инструкции. Если в сборке будет отсутствовать какой-либо класс, программа об этом сообщит. Если в сборке более одного подходящего класса, программа предложит выбрать из списка. Если вы реализовывали класс FileLoader, то программа предложит выбрать текстовый файл для загрузки состояния.

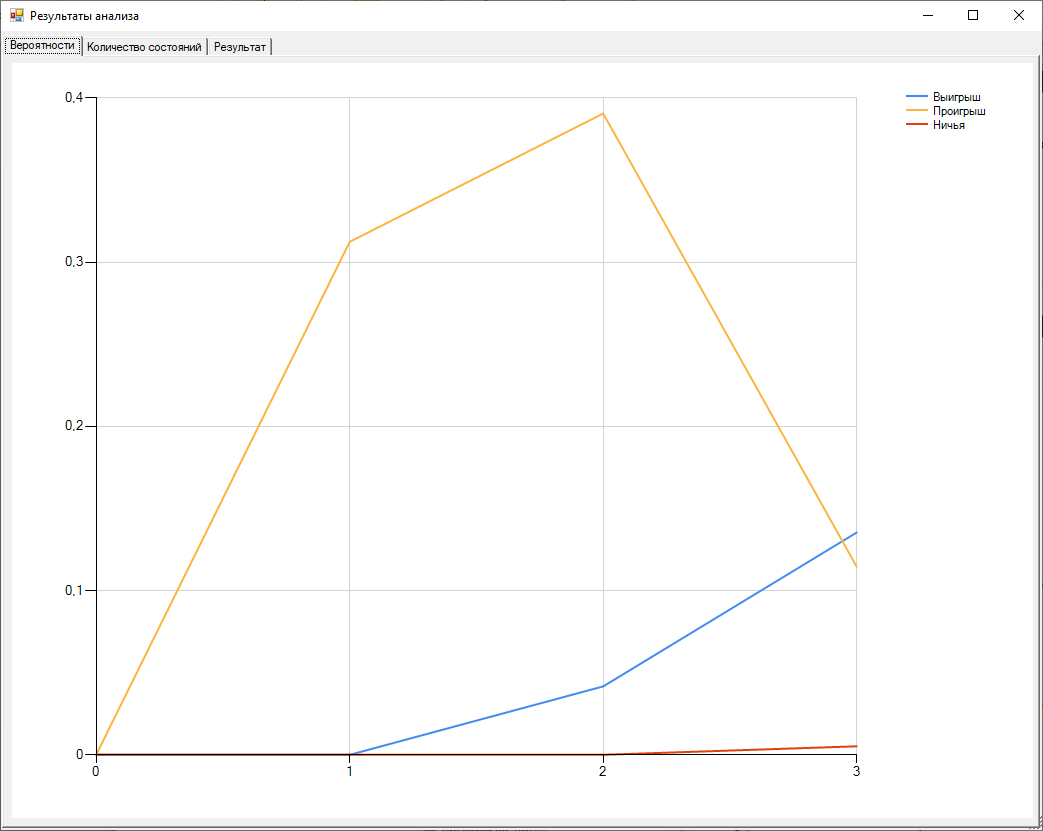
Программа-симулятор



После загрузки данных вам нужно будет просто нажимать на любую клавишу для вывода следующей итерации.

Программа-анализатор





После загрузки данных нажмите любую клавишу, и программа начнет расчеты. После завершения расчетов программа выведет Win-окно с графиками и результатами. Проанализируйте окно, закройте его, и консольное окно предложит вам завершить анализ. Нажимаете любую клавишу.