Design Document
DD1363
Group 20

Per Frost
Marcus Lng
Markus Thurlin
Christopher Engelbrektsson
Christer Hedberg

1. Introduction

Purpose, scope and intended audience
This documents purpose is to provide the design of the system for the programmers to implement. It is based on previously defined requirements. It covers the classes of the system together with their methods and relations to eachother. After reading this document one should be able to make an implementation plan.

This document is intended to be used by project leaders and programmers who will use it to plan and create the system.

Version and reference
This is version 1.0 of the design document for version 1.0 of the system.

Previous documentation is the Requirements Document version 1.0

Briefly on game theory
Game theory is a branch of applied mathematics. It studies the interactions between players or agents in a certain situation and is hence often used for economic purposes. Depending on the type of game or situation the agents will act differently to achieve their goal, a common goal in the economic context would be to maximise profit. The concept of finite game means that there is a definite beginning and end of the game as opposed to infinite games where these are unknown and there is a constant goal to keep playing. Our software will focus only on finite games. A solution concept to a game is a condition that determines an equilibrium of a game or the predicted outcome of the game, the strategy that will be used by the players. An equilibrium is when competing forces are balanced and in general is of course something one does not wish to change from. The Nash equilibrium is a solution concept of a game with multiple players in which the agents individually have chosen a strategy and have nothing to gain by changing it. Subgame perfect equilibrium further builds on this and refers to a part of a larger game and their behaviour for this part represents Nash equilibrium. Another solution concept is correlated equilibria which is the distribution in a game where strategies are chosen at random by this distribution and no player wants to change from this strategy.

Glossary of terms
Strategy
A strategy is a way of deciding which moves to choose in a game.
Solution concept
A solution concept is a class of combinations of strategies that can be considered reasonable.
Information set
An information set is a set of games states between the player to move in them cannot distinguish.
Transition between game states
A transition between game states is move by a player.

Player
A player is an agent who makes moves in the game.
Game tree
A game tree is essentially the same thing as an extensive form game in which players make moves and progress through game states until the reach a game state in which noone is to move.
Normal form game
A normal form game is a game in which each player chooses their moves simultaneously.
Mixed extension of game
The mixed extension of a game is game in which the players chooses the probabilities with which the moves in the game of which the mixed extension is an extension and in which the payoff is the expected payoff.
Nash equilibrium
A Nash equilibrium is a set of strategies from which it is not advantageous to deviate lest others also deviate.
Standard input
Standard input is a file from through which a program can read input to it. Terminals and terminal emulators typically write keyboard input to them to this file.
Standard output
Standard output is a file to which a program can write output. What is written to this file is commonly displayed by a terminal or terminal emulator.

Abstract
This document describes in some detail how the implementation of the system is to be done. This includes all the classes and data handling for the game theory calculation and the design of the Game tree editor at various levels of design.

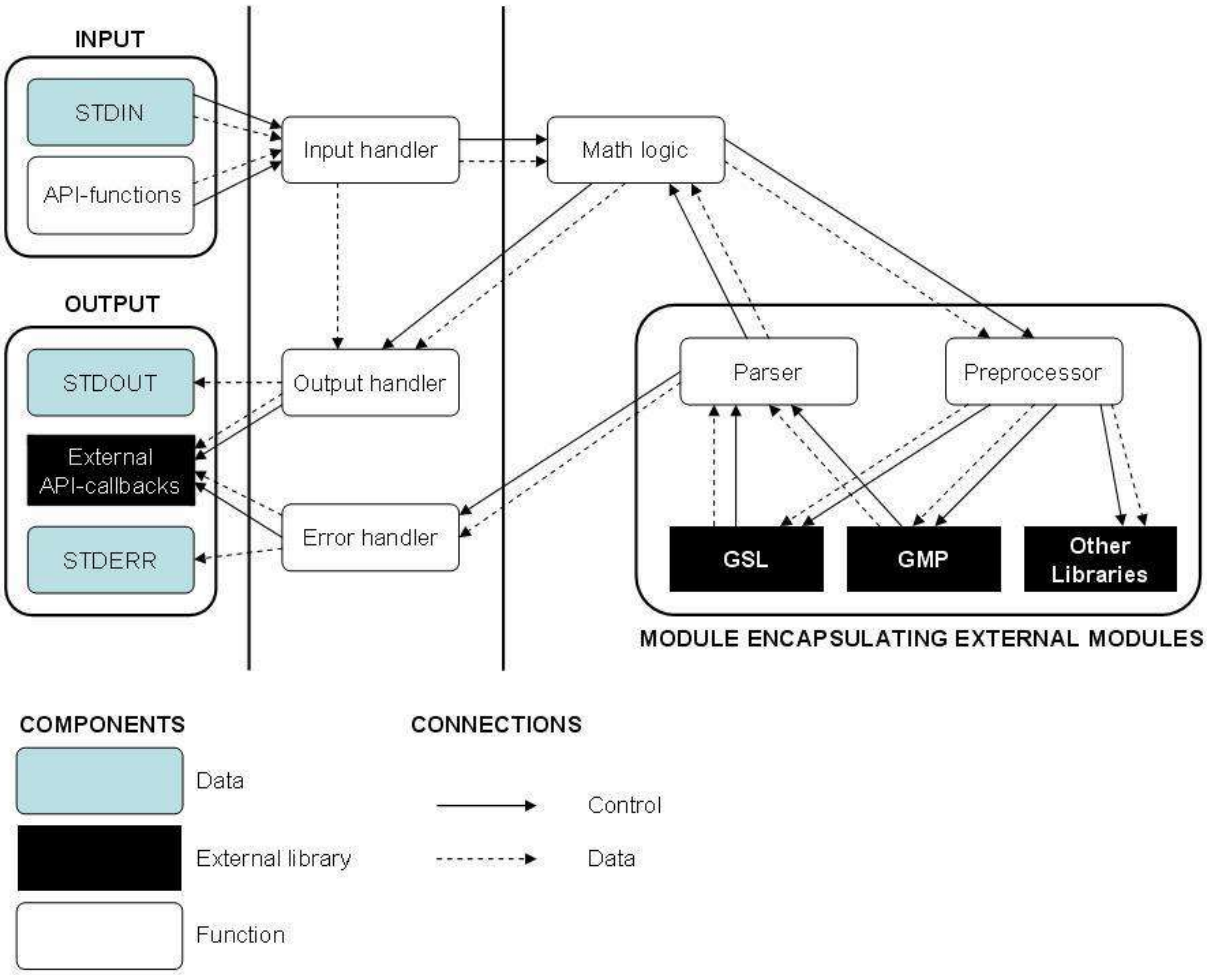
2. System Overview

2.1 General description

The main uses of the system are the construction and editing of representations of finite games, the calculation of subgame perfect equilibria, Nash equilibria, and the development of other computer programs that need to make use of optimal strategies in finite games. A graphical frontend, the game tree editor, will be provided to simplify the creation of game representations. The rest of the system is designed to handle game data and apply the game theory as described in the functionality documents.

The goal is the make a working, flexible system that may be used not only with our game tree editor but by several applications that may have use of our solutions.

2.2 Overall Architecture Description



The general architecture of the program is here presented as a box- & line diagram detailing the different components that it is composed of. The components have a certain color based on what type of component they are: Cyan for data components, black for external library components, and white for function components.

The straight lines between these components represent the control flow and the arrow at the tip of these lines show how control is passed on from one component to another. The dotted lines between these components represent the data flow between the components and the arrow at the tip of these lines show how data is passed on from one component to another.

The two vertical lines in the figure divide the system into three different layers: the first layer (on the left) is the *external layer*, where input is specified and output is presented. The second layer (in the middle) represents the *communication layer* where input, output and errors are handled. The third layer (on the right) is the *core layer*, where all the calculations are processed.

2.3 Detailed architecture

The input handler reads input either from standard input or from external API-functions, determining its format and preprocessing it for the math

logic, passing the determined output format to the output handler and passing control to the math logic. If the input is not correct it passes control and the error is passes to the error handler, but always passes the output mode (to the external API-callbacks or to standard output) to the error handler.

The math logic processes input, calculating the specified equilibria by means of reductions to external libraries accessed by an encapsulation of them, passing the data for the calculations to the encapsulation and once the calculations have completed passes the computed equilibria to the output handler.

The output handler, having received the output format from the input handler and the control handler to the output handler either passes control and data on the specified format to the external API-callbacks, or to standard output.

The error handler recieves control from the input handler or from the encapsulation of of the external math libraries used by the math logic and passes control and the error either to the external API-callbacks or writes the error to standard error.

The encapsulation of the external libraries consists of a preprocessor which receives control and data, preprocessing it and passing it to GMP, GSL, or another library which in turn pass control to a parser, which either passes control and data regarding the error to the error handler or passes data and control to the module from which it received data and control.

External API-callbacks are functions in other programs that we call in order to communicate the result.

CRC Cards

The following Class Responsibility Collaborator (CRC; described by Beck and Cunningham, 1989) cards illustrate how the responsibilities of various tasks are distributed throughout the system.

Input handler
Responsibilities:
-Determine input to pass on to correct math logic function.
Collaborators:
-Math logic

Math logic
Responsibilities:
-Receive data and calculate appropriate results by use of external libraries and forward these to output handler.
Collaborators:
-Input handler
-Module encapsulating external modules
-Output handler

Error handler
Responsibilities:

-Generate appropriate message if an error were to occur.
Collaborators:
-Module encapsulating external modules

Module encapsulating external modules
Responsibilities:
-Make appropriate calls to external libraries and handle the flow between these and math logic function.
Collaborators:
-Math logic
-Error handler

Output handler
Responsibilities:
-Return requested results.
Collaborators:
-Math logic

3. Design Considerations

3.1 Assumptions and Dependencies

The system itself is complicated and the applied game theory mathematics it not something most people know much about. We are assuming that the end-users of the system have enough knowledge of game theory to be able to correctly define games and be able to interpret the results they receive, however the knowledge required to interpret these are less than those required to make any of the calculations. It is also assumed that the system will run on adequate hardware, while the minimum requirements are not high it will be assumed that the end user computer will be equilvalent to the computers in the Magenta computer lab at KTH.

3.2General Constraints

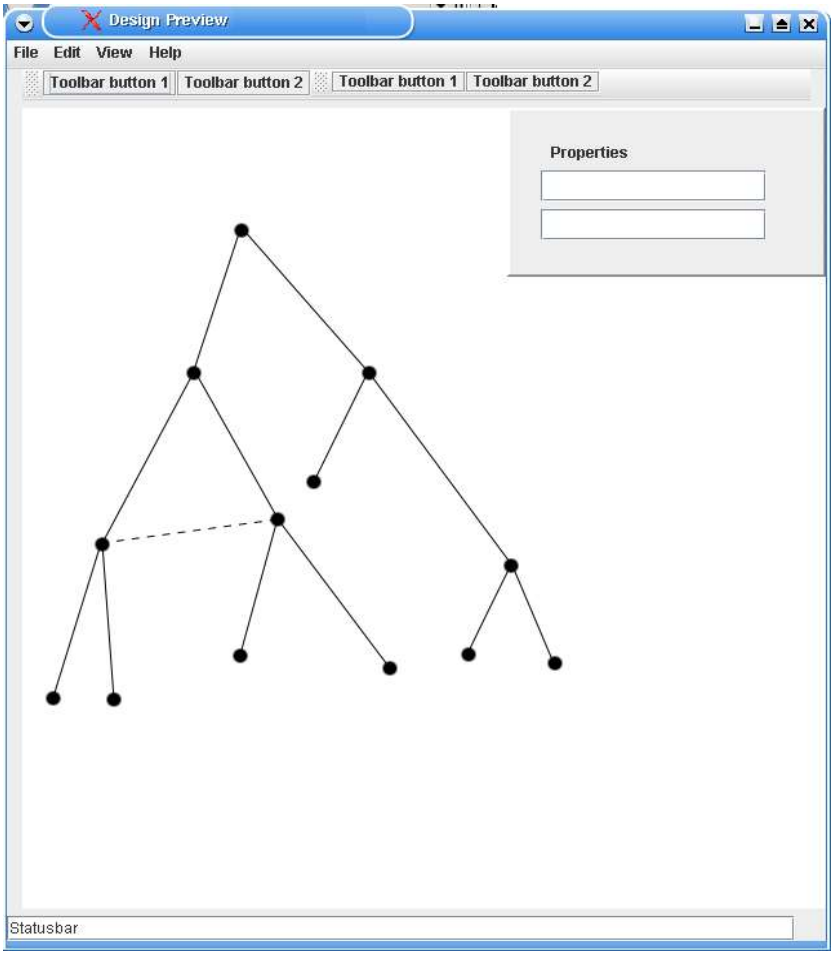
Due to the difficulties of specifying infinite games, and also due to the difficulty of solving them no effort will be made to implement support for these in this system.

4. Graphical User Interface

General Overview of the GUI

The graphical user interface that the user will be working with will be the game tree editor. This will consist of a main working window where the user will use the provided tools to create a graphical representation of the game tree. These tools be will available in a toolbox.

To assign values to the various elements in the representation there will be a property box provided which will contain fields for the values that may be relevant to the element. After the user has created a game tree in the editor and specified its values, he can then select what type of equilibria he wishes to calculate. The calculated equilibria will then be presented in a presentation box in the appropriate format. The user can then make changes to the game tree and easily update this presentation box. The boxes will be movable so that the user can modify the user interface to fit his needs.



GUI Screenshot (where the toolbar buttons represent the panels detailed below)

Detailed overview of the components of the GUI

The graphical user interface consists of a main window with a drop down menu at the top, an editing area and several panels that may be moved about by the user at will and may be docked against edges of the window and result windows.

Document elements are results, points, lines, dotted lines, rows, columns, tables, and table elements. They are displayed in the editing area. They can be selected individually with a pointer, and sets of them can be selected by holding shift and selecting those that are to be added to the current set with the mouse as when individual elements are to be selected. Document elements can be dragged with the pointer, and Result can be displayed in separate windows, it can also be dragged out of the editing area and fit to a box.

How Results are displayed is decided by what type of results they are (i.e. based on the results of the specific types of calculations that have been made), and it is inappropriate to detail it in the description of the user interface.

Menu choices

The drop down menu has the entries File, Edit, View and Help. If an entry on any of these menus is clicked submenus are displayed.

Submenus for the different menu choices:
In the case of File: New, Open, Save, Save as, Close and Edit.
In the case of Edit: Undo, Do, Cut, Copy, Fit box, Delete, Rescale and Pan.
In the case of View: Zoom in, Zoom out and Fit screen.
In the case of Help: Documentation.

Submenu choices for File:
If New is selected a new editing window is to be displayed.
If Open is selected a standard 'open'-dialog will appear.
If Save is selected a standard 'save'-dialog will appear.
If Save as is selected a standard 'save as'-dialog will appear.
If Close is selected the current file is to be closed.
If Exit is selected the program is to exit completely.

Submenu choices for Edit:
If Undo is selected any action that has changed the state of that which is displayed in the editing area is to be reverted.
If Redo is selected an Undo is to be undone in the same manner as selecting Undo would have undone a regular action.
If Cut is selected the set of document elements that are selected are to be removed from the document and placed in the paste buffer, the internal relations between the elements retained.
If Copy is selected the set of selected objects in the editing area are treated in the same

manner as in the case of Cut, but with the elements not being removed.
If Delete is selected the set of document elements that are selected in the editing area are to be removed.
If Fit box is selected the user selects a set in the editing area with the pointer and fit it to a box outside the editing area.
If Rescale is selected the set of selected elements in the editing area are rescaled to a scale indicated by the user by moving the pointer.
If Pan is selected the editing area may be repositioned by dragging it.

Submenu choices for View:
If Zoom in is selected the editing area is zoomed in on.
If Zoom out is selected the editing area is zoomed out of.
If Fit screen is selected the editing area is fitted to screen.

Submenu choices for Help:
If Documentation is selected an HTML-document containing documentation is opened in whatever web-browser there is on the system.

Panels

There is a *property panel* with a probability box and utility box, the first in which that a value is entered indicates that the same value is to be assigned to the currently selected line and the second one which into which that a value is entered indicates that a utility is to be assigned to a currently selected node.

There is a *tool panel*, from which tools to be made active are selected. The things that can be selected from the tool panel are: Point, Line, Dashed Line, Rotate selected, Delete, Move, Row and Column.

Panel choices for the tool panel:
If Point is selected it is indicated that points are to be created by clicks in the editing area.
If Line is selected it is indicated that lines are to be created in the editing area.
If dashed line is selected it is indicated that dashed lines are to be created between the set of selected points.
If Rotate selected is selected the selected elements are to be rotated according to how the user moves the mouse pointer.
If Delete is selected the currently selected document elements are to be removed.
If Move is selected it is indicated that document elements are to be moved about in the editing area when selected moved about.
If Row or Column is selected, Row or Column document elements are to be created.

There is also a *view panel*, consisting of Zoom, which functions as the zoom element in the menu, Pan which functions as the pan element in the editing menu, Fit to screen, which fits the document element in the editing area to it, Fit selected to box, which fits the selected document elements in the editing area to a box.

There is also a *calculate panel*, consisting of the elements Calculate, OnChange, an InNewWindow/InWindow-toggle, and Annote Game tree toggle.
Elements for the calculate panel:
The element Calculate is a drop down menu from which things to be calculated can be selected.
The OnChange box is a box into which results can be dragged, that they are dragged there indicating that they are to be automatically recalculated upon any change in that from which they are calculated.
The InNewWindow/InWindowMode-toggle is such that if the toggle is in the state 'InNewWindow' Results from newly done calculations are to be displayed in new windows as opposed to otherwise in the editing area. All Result windows must be saveable.
If the Annote game tree toggle is active a game is annoted with the subgame perfect equilibrium if it is calculated instead of being displayed separately.

At the very bottom of the window there is to be an *error bar* in which reasons why things in the editing area cannot be created are detailed.

5.1 Introduction

The following sections give a more detailed overview of all the classes of the system providing descriptions and diagrams as labeled.

5.2 Class Resposibility Collaborator Cards

InputStream

Responsibilities

The input stream class inherits from the input stream class provided by the C++ standard library. The class is responsible for encapsulating an input stream, in particular its creation input stream, that of reading data from it, the maintaining of its state and that of returning errors when errors occur. It is constructed from the input stream of the standard library, normal form games can be read from it, extensive form games can be read from it, solution types can be read from it, and the input type and the output type are members of it.

Collaborators

The input stream of the standard library, NormalFormGame, ExtensiveFormGame, SolutionType, InputType, OutputType, ParseError.

OutputStream

Responsibilities

The output stream class inherits from of the ouput stream class in the C++ standard library and maintains except for the output stream from which it inherits from an output type. The class is responsible for encapsulating an output stream, in particular creating an It is constructed from an output stream and an output type and solutions can be written to it.

Collaborators

The output stream class of the standard library, OutputType, Solution.

InputType

Responsibilities

The input type is an enumeration of supported input types and has no responsibilities but those of storing a supported input type.

Collaborators

InputStream.

OutputType

Responsibilities

The output type is an enumeration of supported output types and has no responsibilities but those of storing a supported output type.

Collaborators

OutputStream, InputStream.

SolutionType

Responsibilities

The solution type is an enumeration of supported solution types and has no responsibilities but those of storing a supported output type.

Collaborators

InputStream, OutputStream.

NormalFormGame

Responsibilities

The normal form game stores a representation of a normal form game, it maintains a player set, a set of strategies a payoff function from the game the cartesian product of the set of strategies to the reals. It provides references to these sets. It is constructed directly from the sets to which it provides references.

Collaborators

InputStream.

ExtensiveFormGame

Responsibilities

The extensive game stores a representation of an extensive game, it supplies references to a set of players, to a set of game states, to a set of moves, to a bijective function from the game states to the moves, to a map from the game states to sets of game states, and from the cartesian product of the set of moves to a cartesian product of payoffs corresponding to the players. It provides references to these, and provides a function that takes a solution type and returns a solution.

Collaborators

InputStream.

LinearProgram

Responsibilities

Linear program maintains a matrix and the transpose of a vector It represents a linear program on canonical form. It provides references to the vector, the matrix and certain functions thereof for the construction of linear program solutions.

Collaborators

LinearProgramSolution.

LinearProgramSolution

Responsibilities

A linear program solution is a vector and inherits from the vector class provided by the C++ standard library, it is constructed from a linear program.

Collaborators

The vector class from the C++ standard library, LinearProgram.

ParseError

Responsibilities

Parse errors represent exceptions and store a string. A routine enables them to be written to the ouput stream standard error.

Collaborators

InputStream.

Solution

Responsibilities

A solution is a base class for solutions that may be written. It is responsible for storing a solution and for being writeable to an output stream, why it provides methods so that the ouput stream can write it on any of the enumerated output formats.

Collaborators

SubgamePerfectEquilibrium, NashEquilibrium.

SubgamePerfectEquilibrium

Responsibilities

Subgame perfect equilibrium inherits from solution and stores a subgame perfect equilibrium. It provides routines for being written to output stream.

Collaborators

OuputStream, Solution.,

NashEquilibrium

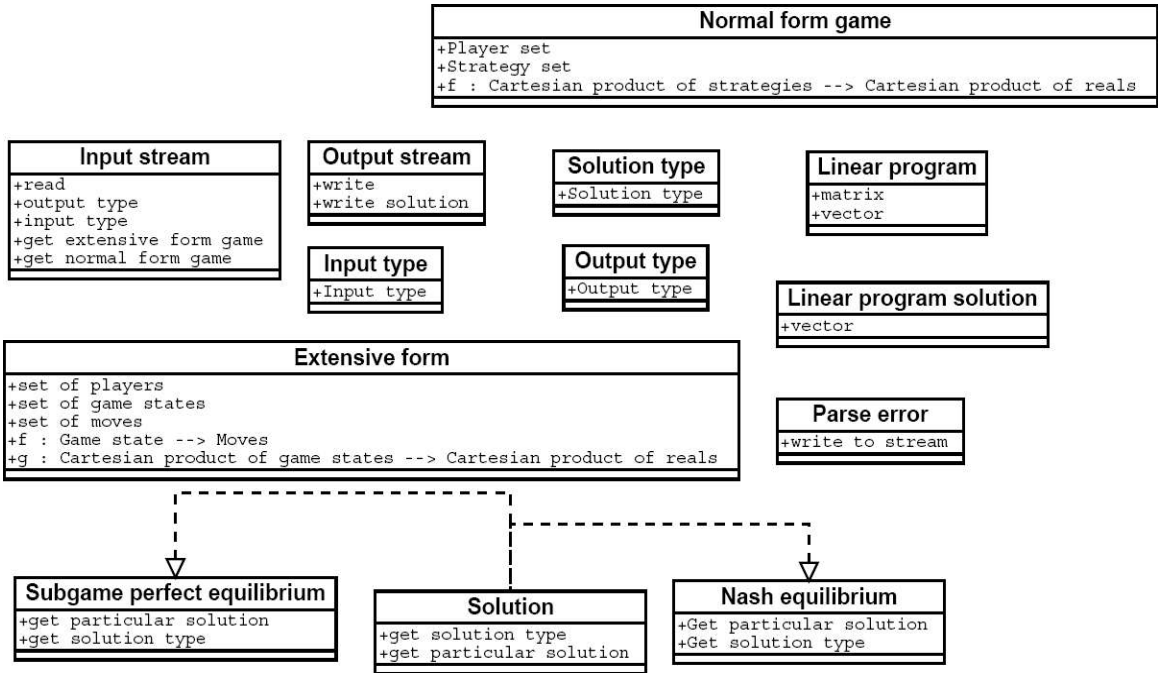
Responsibilities

Nash equilibrium inherits from from solution and stores a nash equilibrium, it provides routines for being written to output stream.

Collaborators

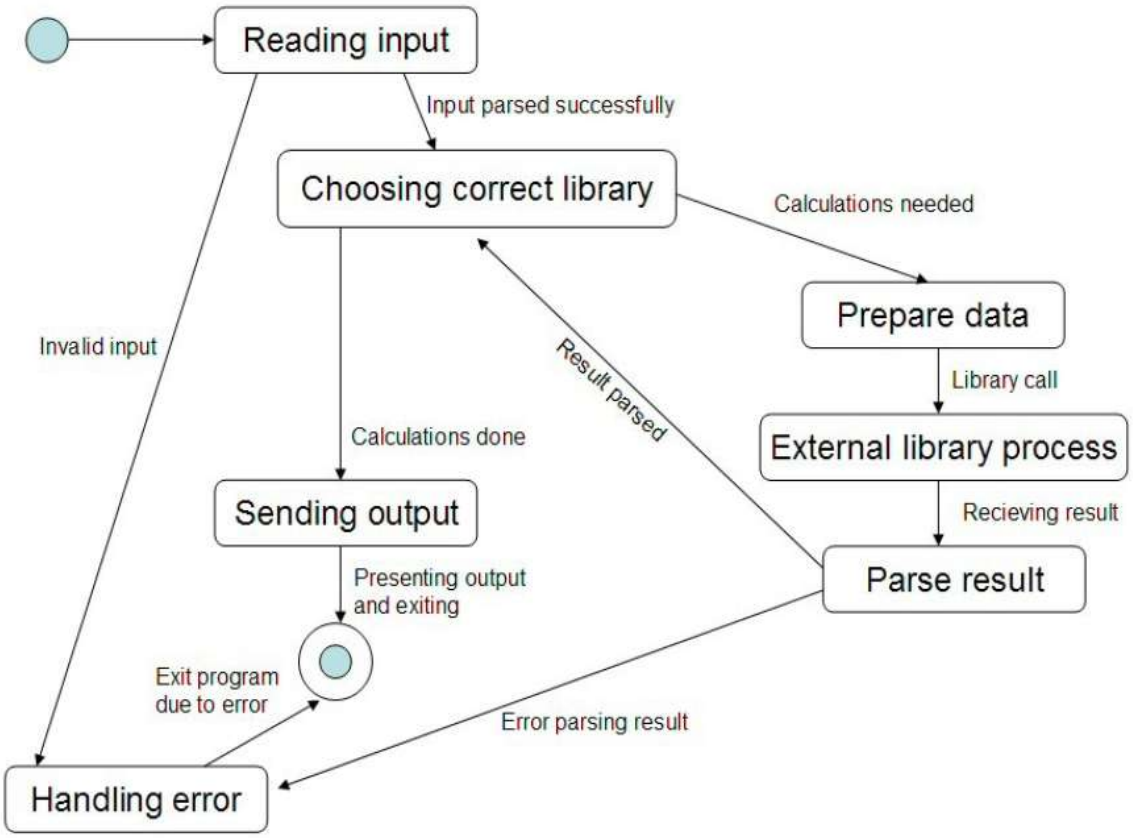
OutputStream, Solution.

5.2 Class diagrams

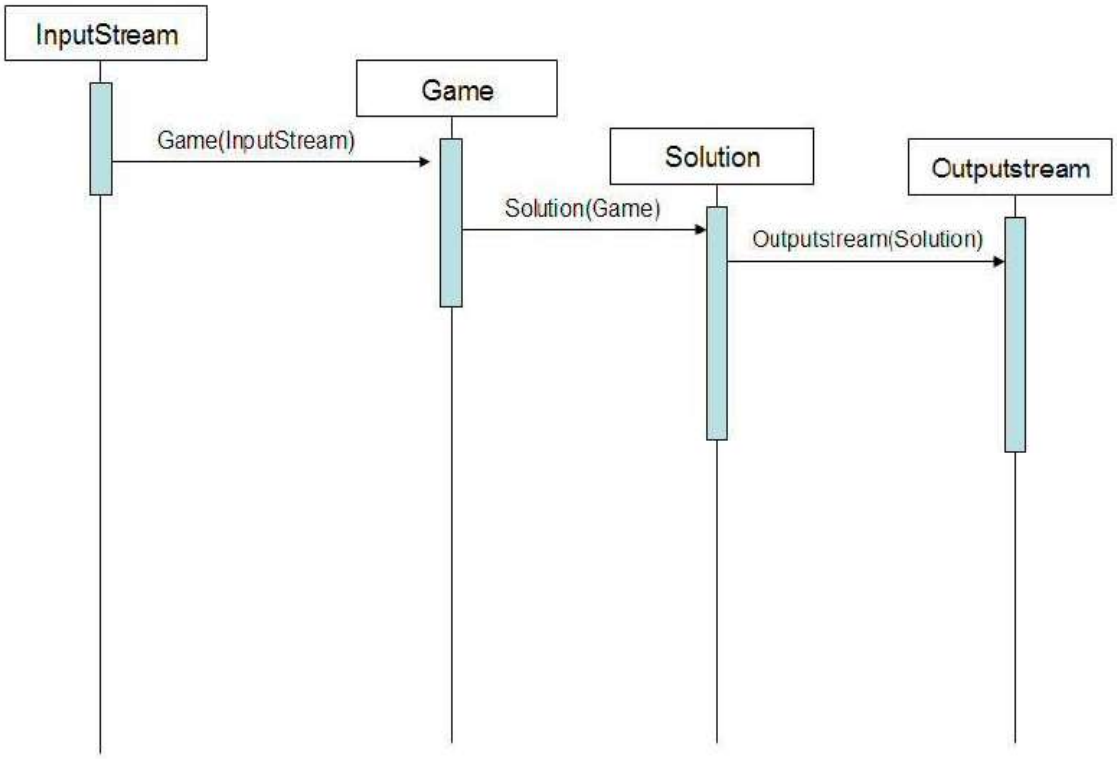


5.3 State Charts

Typical usage



5.4 Interaction Diagrams



5.5 Detailed Design

InputStream

Method name: InputStream
Parameters: A reference to an istream
Return value: Inapplicable
Description: Constructs an InputStream from an istream object
Pre-conditions: The istream object contains a representation of an InputStream.
Validity checks, errors, and other anomalous situations:
Post-conditions: None
Called by:
Calls: None
Accesses: local variables of InputStream

Method name: InputStream
Parameters: A pointer to a callback function
Return value: Inapplicable
Description: Constructs an InputStream from a callback function.
Pre-conditions: The istream object contains a representation of an InputStream.
Validity checks, errors, and other anomalous situations:
Post-conditions: None
Called by:
Calls: None
Accesses: local variables of InputStream

Method name: getInputType
Parameters: None
Return value: An element defining an allowed input.
Description: Returns the input type specified in an istream object.
Pre-conditions: The istream object contains a representation of an InputStream.
Validity checks, errors, and other anomalous situations:
Post-conditions: None
Called by:
Calls: None
Accesses: local variables of InputStream

Method name: getOutputType
Parameters: None
Return value: An element defining an allowed output.
Description: Returns the output type specified in an istream object.
Pre-conditions: The istream object contains a representation of an InputStream.
Validity checks, errors, and other anomalous situations:
Post-conditions: None
Called by: The constructor of NormalFormGame and the constructor of ExtensiveFormGame.
Calls: None
Accesses: local variables of InputStream

Method name: getExtensiveFormGame
Parameters: none
Return value: an object of type ExtensiveGame
Description: Returns an ExtensiveGame constructed from the InputStream

Pre-conditions: The getInputType method returns a value consistent of that of a ExtensiveGame.
Validity checks, errors, and other anomalous situations:
Post-conditions: None
Called by:
Calls: The constructor of ExtensiveFormGame
Accesses: None

Method name: getNormalFormGame
Parameters: none
Return value: an object of type NormalGame
Description: Returns an NormalGame constructed from the InputStream
Pre-conditions: The getInputType method returns a value consistent of that of a NormalGame.
Validity checks, errors, and other anomalous situations:
Post-conditions: None
Called by:
Calls: The constructor of NormalFormGame
Accesses: None

OutputStream

Method name: OutputStream
Parameters: ostream&
Return value: Inapplicable
Description: Constructs an OuputStream from an ostream object
Pre-conditions: The ostream object contains a representation of an OutputStream.
Validity checks, errors, and other anomalous situations:
Post-conditions: None
Called by:
Calls: None
Accesses:

Method name: OutputStream
Parameters: A pointer to a callback function
Return value: Inapplicable
Description: Constructs an OuputStream from a callback function.
Pre-conditions: The ostream object contains a representation of an OutputStream.
Validity checks, errors, and other anomalous situations:
Post-conditions: None
Called by:
Calls: None
Accesses: None

ExtensiveFormGame

Method name: ExtensiveFormGame
Parameters: references to a set of players, to a set of gamestates, to a set of moves, to a bijective function from the game states to the moves, to a map from the game states to sets of game states, and from the cartesian product of the set of moves to a cartesian product of payoffs corresponding to the players.
Return value: None
Description: Stores a representation of an extensive game.
Pre-conditions: None

Validity checks, errors, and other anomalous situations:
Post-conditions: None
Called by: getExtensiveFormGame
Calls: None
Accesses: None

NormalFormGame

Method name: NormalFormGame
Parameters: references to a set of players, to a set of strategies, to a payoff function from the games cartesian product of the set of strategies to the reals.
Return value: None
Description: Stores a representation of a normal game.
Pre-conditions: None
Validity checks, errors, and other anomalous situations:
Post-conditions: None
Called by: getNormalFormGame
Calls: None
Accesses: None

LinearProgram

Method name: LinearProgram
Parameters: A vector and a matrix
Return value: None
Description: Represents a linear program on canonical form.
Pre-conditions: None
Validity checks, errors, and other anomalous situations:
Post-conditions: None
Called by: Arbitrary classes in the math logic package
Calls: None
Accesses: None

Method name: getLinearProgramVector
Parameters: none
Return value: vector
Description: Returns the vector maintained
Pre-conditions: LinearProgram has been constructed.
Validity checks, errors, and other anomalous situations:
Post-conditions: None
Called by: LinearProgramSolution
Calls: None
Accesses: local vector variable

Method name: getLinearProgramMatrix
Parameters: none
Return value: matrix
Description: Returns the matrix maintained
Pre-conditions: LinearProgram has been constructed.
Validity checks, errors, and other anomalous situations:
Post-conditions: None
Called by: LinearProgramSolution

Calls: None
Accesses: local matrix variable

LinearProgramSolution

Method name: LinearProgramSolution
Parameters: LinearProgram object
Return value: None
Description: Solves linear program and stores solution in vector
Pre-conditions: LinearProgram has been constructed.
Validity checks, errors, and other anomalous situations:
Post-conditions: Vector is stored.
Called by: Arbitrary classes in the math logic package
Calls: getLinearProgramMatrix, getLinearProgramVector
Accesses: LinearProgram variables

Method name: getLinearProgramSolutionVector
Parameters: None
Return value: vector
Description: Returns the solution vector.
Pre-conditions: Vector has been stored
Validity checks, errors, and other anomalous situations:
Post-conditions: None
Called by: Arbitrary classes in the math logic package
Calls: None
Accesses: Local vector variable

Solution

Method name: write
Parameters: A reference to an OutputStream
Return value: Boolean
Description: Writes the solution to the InputStream.
Pre-conditions: Inapplicable as Solution is pure virtual.
Validity checks, errors, and other anomalous situations:
If and only if the procedure exits successfully it will return true.
Post-conditions: None
Called by: None
Calls: Inapplicable
Accesses: Inapplicable

NashEquilibrium

Method name: NashEquilibrium
Parameters: NormalFormGame
Return value: Inapplicable
Description: Constructs a Nash Equilibrium from a normal form game
Pre-conditions: None
Validity checks, errors, and other anomalous situations: None
Post-conditions: None
Called by: None

Calls: The constructor of LinearProgramSolution, the accessors of LinearProgramSolution and the constructor of LinearProgram
Accesses: Local variables of NashEquilibrium.

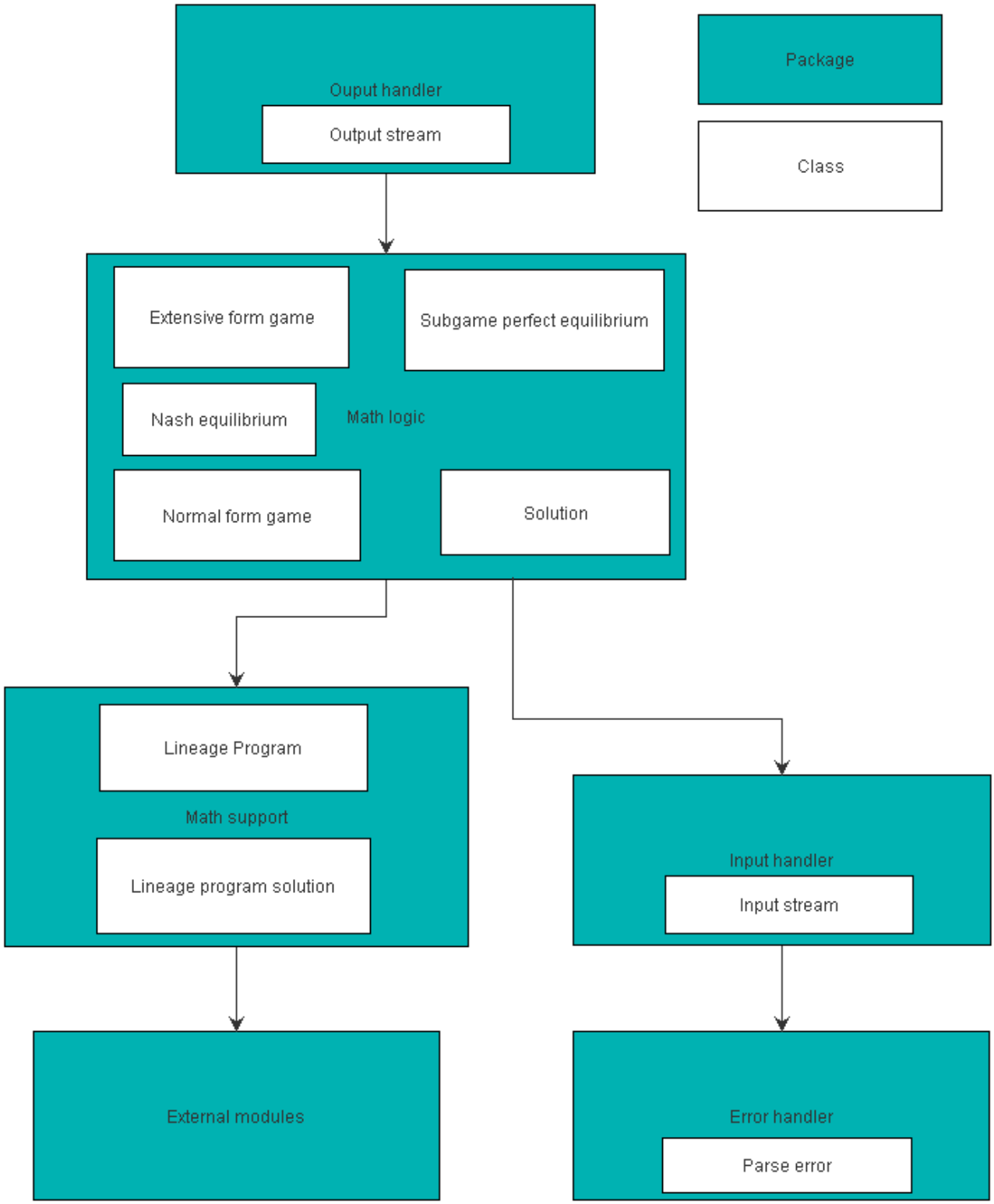
Method name: write
Parameters: A reference to an OutputStream
Return value: Boolean
Description: Writes the solution to the OutputStream.
Pre-conditions: Inapplicable as Solution is pure virtual.
Validity checks, errors, and other anomalous situations:
If and only if the procedure exits successfully it will return true.
Post-conditions: None
Called by: None
Calls: The inherited operators of OutputStream
Accesses: None

SubgamePerfectEquilibrium

Method name: SubgamePerfectEquilibrium
Parameters: ExtenisveFormGame
Return value: Inapplicable
Description: Constructs a Nash Equilibrium from an extensive form game
Pre-conditions: None
Validity checks, errors, and other anomalous situations: None
Post-conditions: None
Called by: None
Calls: The constructor of LinearProgramSolution, the accessors of LinearProgramSolution and the constructor of LinearProgram
Accesses: Local variables of SubgamePerfectEquilibrium.

Method name: write
Parameters: A reference to an OutputStream
Return value: Boolean
Description: Writes the solution to the OutputStream.
Pre-conditions: Inapplicable as Solution is pure virtual.
Validity checks, errors, and other anomalous situations:
If and only if the procedure exits successfully it will return true.
Post-conditions: None
Called by: None
Calls: The inherited operators of OutputStream
Accesses: None

5.6 Package diagram



6. Functional Test Cases

Define preferences for game states/payoffs for game states

Description

The system shall provide a method for the user to define preferences/payoffs for game states. This shall be tested by having some person associate a player with some game state.

Reference

Requirements Document, Section 6.1, Part **Define preferences for game states/payoffs for game states**

Inputs

A game state consists of atleast a payoff value.

Expected output

The expected output is a node that has atleast a value representing its payoff. For further definition of the game state see 'Associate players with game states'. The game state will be displayed as a node.

Step by step procedure

1. The user has the program open. 2. The user clicks the create game state button. 3. The user chooses a value for the gamestate. 4. The game state will be created and will be represented as a node on the screen.

Define information sets

Description

The system shall provide a method for the user to define information sets. This shall be tested by having any person define an information set.

Reference

Requirements Document, Section 6.1, Part **Define information sets**

Inputs

The information set is defined by a set of game states. The user does this by selecting nodes on the screen in the game tree editing tool.

Expected output

The expected output is a set of nodes as selected by the user. The result will be visible to the user in the graphical representation by connection with dashed lines between these nodes.

Step by step procedure

1. The user has defined gamestates on the screen. 2. The user chooses the tool for creating information sets. 3. The user selects the nodes he desires to be included in the set. 4. The user can observe the visible output of the changes to see if it is as desired.

Define player sets

Description

The system shall provide a method for the user to define player sets. This shall be tested by having some person define a player set.

Reference

Requirements Document, Section 6.1, Part **Define player sets**

Inputs

The player set consists of the players in a game, each player is defined simply by a name.

Expected output

A set of players who are included in a game.

Step by step procedure

1. The user clicks add player option. 2. The user enters a name for the player. 3. The player is added to the set or if there is no set a set is created for the game and the player is added. 4. The user will be able to see the player set as a list of players in a game including the added one.

Define transition between elements

Description

The system shall provide a method for the user to define transitions between elements. This shall be tested by having some person define a transition between some pair of two elements.

Reference

Requirements Document, Section 6.1, Part **Define transition between elements**

Inputs

The transitions between elements are simply the moves of the game, a tranistion from one gamestate to another. The input to this would be two nodes and the direction of the allowed move would be defined by the order in which the nodes are selected (as from 'first node' to 'second node').

Expected output

The expected output is a set of two gamestates. To the user this will be visibly observable by a connection line between the two gamestates with an arrow in the direction of the allowed move.

Step by step procedure

1. The user has 2 or more defined game states on the screen. 2. The user selects the tool for adding a transition. 3. The user selects two nodes that represent game states in the order he desired. 4. A transition will be defined between the two nodes and it will be displayed as an arrowed line between the nodes in the direction allowed.

Associate players with game states

Description

The system shall provide a method for the user to associate players with transitions. This shall be tested by having some person associate a player with some game state.

Reference

Requirements Document, Section 6.1, Part **Associate players with transitions**

Inputs

Associating a player with a game state requires a defined game state and a defined player.

Expected output

The expected output is a tuple consisting of a game state and a player. The association will be visible to the user.

Step by step procedure

1. The user has atleast one defined game state and atleast one defined player. 2. The user selects the game state and is able to see the value box. 3. The user can set the player parameter to any player that is defined in the game. 4. The user will see which player is associated with each transition by a lable at the game states node.

Tree-editing tool

Description

The system shall allow for taking graphical input and converting into a game tree. The tool will provide methods for creating the graphical input and use the required defined elements to create a game tree. The tree shall constantly be displayed regardless of its completeness. The user shall be able to store this tree to disk.

Reference

Requirements Document, Section 6.1, Part **Tree-editing tool**

Inputs

The game tree may consist of all elements described above.

Expected output

The expected output is a complete gametree in a clear visible format that is in a state it is ready to have the desired calculations applied.

Step by step procedure

1. The user defines all game states, tranitions, players and associates them as desired in a tree format. 2. The user clicks a button to 'complete the game'. 3. The system checks the suggested game tree and if anything is incomplete the user is informed and moved back to step one where he can complete the

tree. 4. The user is informed that the game is properly defined and the game is displayed with the options to perform calculations.

Calculate pure Nash equilibrium

Description

The system shall, given a valid input, calculate the pure Nash equilibria and return it in a requested output form. The user shall have the option of storing the result to disk.

Reference

Requirements Document, Section 6.1, Part **Calculate pure Nash equilibrium**

Inputs

A correctly defined game.

Expected output

The expected output will be a set of pure nash equilibria and will be displayed to the user as a set of sets of transitions.

Step by step procedure

1. The user has a correctly defined game. 2. The user clicks the option to the calculate pure Nash equilibrium. 3. The system will process the game and calculate its pure nash equilibria. 4. The output will be displayed to the user.

Calculate Nash equilibria in the mixed extension of a game (mixed Nash equilibria

Description

The system shall, given a valid input, calculate the mixed Nash equilibria and return it as requested output form. The user shall have the option of storing the result to disk.

Reference

Requirements Document, Section 6.1, Part **Calculate Nash equilibria in the mixed extension of a game (mixed Nash equilibria**

Inputs

A correctly defined game. For this calculation the user may define probability values for the transitions between game states.

Expected output

The expected output will be a set of mixed Nash equilibria and will be displayed to the user as a set of sets of transitions.

Step by step procedure

1. The user has a correctly defined game. 2. The user clicks the option to calculate the mixed Nash equilibrium. 3. The system will process the game and calculate its mixed Nash equilibria. 4. The output will be displayed to the user.

Calculate correlated equilibria

Description

The system shall, given a valid input, calculate the correlated equilibria and return it as requested output form.

Reference

Requirements Document, Section 6.1, Part **Calculate correlated equilibria**

Inputs

A correctly defined game.

Expected output

The expected output will be a set of correlated equilibia and will be displayed to the user as a set of sets of transitions.

Step by step procedure

1. The user has a correctly defined game. 2. The user clicks the option to calculate the correlated equilbrium. 3. The system will process the game and calculate its correlated equilibria. 4. The output will be displayed to the user.

Calculate sub game perfect equilibria

Description

The system shall, given a valid input, calculate the sub game perfect equilibria and return it as requested output form. The user shall have the option of storing the result to disk.

Reference

Requirements Document, Section 6.1, Part **Calculate sub game perfect equilibria**

Inputs

A correctly defined game.

Expected output

The expected outut will be a set of subgame perfect equilibria and will be displayed to the user as a set of sets of transitions.

Step by step procedure

1. The user has a correctly defined game. 2. The user clicks the option to calculate the sub game perfect equilibrium. 3. The system will process the game and calculate its sub game perfect equilibria. 4. The output will be displayed to the user.