Gambit Project
Conversion Programs Final Report

Prepared by Karen Bletzer
Last updated on September 4, 2011

Sponsored by Google Summer of Code 2011

Table of Contents

Introduction

The Gambit Project seeks to create and maintain software tools for the exploration of game theory. The current toolbox includes Gambit, which includes both a graphical interface for building games, as well as command line utilities. The toolbox also includes Game Theory Explorer (gte), a Java/Flex application currently in beta release. Gte supports the building of extensive games as well as the calculation of equilibria via the Lemke-Howson algorithm.

Gambit supports several file formats. In this project we will discuss the efg and nfg formats in particular. However, note that there are other formats, such as gbt, which are also supported.

Gte currently supports an XML representation of extensive form games, including the reading and writing of XML files. This existing XML specification will be extended to support strategic (normal) form games, as well as to support parameters relevant to the creation of LaTeX representations of the strategic form games. Additionally, the extensive form game XML representation will be updated to include game tree data elements and features that may be supported by gte in the future.

In this document we will discuss existing file formats, the proposed new XML structure, and the implementation of conversion utilities to transform game files.

Problem Background

In this section we will cover background concepts used in the remainder of this document, including the XML proposal, game formats, and conversion approaches.

Extensive Games

Game trees can be used to represent non-cooperative games. A game tree, sometimes described as the extensive form representation of a game, describes how a game evolves over time, as each player makes a decision in response to the given game state and possible outcomes. Game trees are composed of nodes, and edges (lines), which connect the nodes. The nodes represent the state of the game at a particular point in time. The edges represent a possible move in the game. Coverage of game trees as well as the strategic games below is based on material by Bernhard von Stengel [1].

Nodes may be classified as terminal nodes, root, or interior nodes. Terminal nodes are leaf nodes; they have no children. Terminal nodes represent possible end states of the game. Players may receive a payoff at a terminal node.

The root node represents the beginning state of the game; the root has no predecessors. Interior nodes are those nodes which are not terminal nodes. The root node is also an interior node.

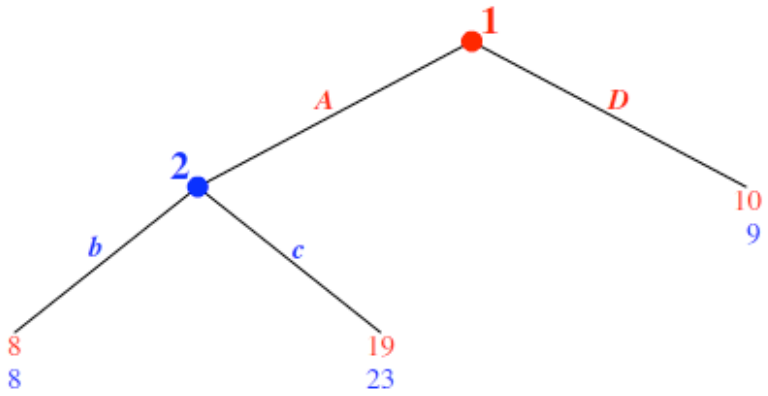


Figure 1. Extensive Game Tree; figure generated via gte tool.

They two types of interior nodes may be classified by the type of the decision that can be made at the node. For chance nodes, the decision is determined by pre-defined probabilities known to the players. At a decision node, sometimes referred to as a player node, a player makes the decision on what move to take next based on the information he/she knows about the game and its outcomes.

Strategic Games

The strategic game form is another way to represent non-cooperative games. Given the set of each player's strategies from the game tree, and the corresponding payoffs, one can construct a table to represent the game. In the table the rows represent the strategies that are available to player I, and the columns represent the strategies that are available to player II. In this treatment of games Player I will be called the row player, and Player II the column player. The payoffs for each player, given the strategies labeled in the rows and columns, are represented in the cells of the table. Player I's payoffs are at the bottom left of the payoff cell. Player II's payoffs are found on the top right of the cell. The strategic game hold data similar to the game tree, but flattened in time. That is, the players choose all of their strategies simultaneously, instead of move by move as in the game tree. Alternatively, the strategic game may be interpreted independently of an extensive game, as a formulation of decisions that truly are simultaneous in time.

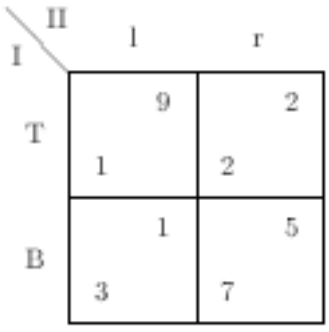


Figure 2. A strategic game between two players; each player has two strategies to choose from.

Gambit

Gambit is a library of game theory software supported and maintained by the Gambit Project [3]. The Gambit tools support the building of extensive and strategic games, as well as the calculation of Nash equilibria and other key metrics. The building of games and some select calculations may be done via the graphical user interface. Other non-graphical tools that calculate equilibria via different algorithms may be accessed via the command line.

gte

The gte program is a hybrid Java/Flex application deployed to the web via the Google App Engine platform. It supports the building of extensive games as well as the calculation of equilibria via the Lemke-Howson algorithm. It is an experimental addition to the Gambit Project's suite of game theory tools.

XML

Extensible Markup Language, more commonly known as XML, allows the creation of data-driven documents that can be easily read and processed by machines. XML documents consist of several basic building blocks which are repeated and arranged to create the document. An XML document's required structure is defined by the authors of the software, document, or interface.

There are several key concepts that are helpful in understanding XML documents and their design. XML, being an extremely popular way of communicating data and information, is covered in a variety of books and websites. A further concise explanation of XML concepts can be found here [4]. The following material is a brief introduction to the subject.

Markup and Content. Text in an XML document is either classified as markup or content. Markup is any information enclosed within the characters "<" and ">". Content is any text that is not classified as markup. For example, in the following XML excerpt *<payoff player="Player 1">*11*</payoff>* the italicized portion is markup, and the remaining portion, the value 11, is content.

Tag. Tags are a subset of markup starting with "<" and ending with ">". Start tags indicate the beginning of an XML element: *<payoff player="Player 1">***.** End tags pair up with the start tags to enclose data: *</payoff>***.** Empty element tags also can be used: *<payoff/>*. The empty element tag is a shorthand for tags with no content, such as *<payoff></payoff>*.

Element. An element is a unit of data that begins with a start tag and is completed by an end tag. An element's content could include text data as in the first example below.

```
<payoff player="Player 1">11</payoff>
```

Alternatively, an element's content could include other elements, as below, where the outcome element includes two payoff elements.

```
<outcome outcomeId="1">
        <payoff player="Player 1">5</payoff>
        <payoff player="Player 2">7</payoff>
</outcome>
```

It's possible that an element's content might be empty. For example, an empty tag for payoff, *<payoff/>*, indicates that no payoff information is available; however that formulation is still an element.

Attribute. Attributes are another category of markup which consist of a name/value pair stored within the "<" and ">" pair. That is, attributes are data stored within the tag itself. In the example below, the "player" information is an attribute with the name "player" and the value "Player 1".

```
<payoff player="Player 1">5</payoff>
```

Attributes are limited to name-value pairs. As such, attributes are limited to one instance per tag. This is in contrast to elements, or content data, which can be expanded and extended in the case of multiples, etc.

File Formats

efg files

The efg file format is one of the formats used by the Gambit tools to store and communicate extensive form games. This format is described in detail in [3]. The efg file defines the game description, the node types, the payoffs at the terminal nodes, and all of the structures in between. The format supports games with multiple players, as well as games with payoffs at either internal or terminal nodes.

The following types of data can be found in an efg file:

- Game Description – A short description of the type of game, typically citing the source or giving a high level description of the game background. For example, "General Bayes game, one stage."
- Node Type – "p", "c" or "t" for player, chance or terminal node
- Player Name and number - The mapping of player number to player name (if available) is done in the header line of the file by associating the first player listed as player 1, and so on. In each node line the player is represented by the player number.
- Node Name – optional name for the node. The delimiting quotation marks are required.
- Iset number – a number identifying the particular iset for the node. The iset numbering is unique for a given player.
- Iset name – an optional name associated to the iset. The delimiting quotation marks are required.

- Allowed moves – a list of moves/actions that lead to the game state as represented by the node's children. The move information can be thought of as belonging to the edges directed away from the node to its children. When the node is a chance node each move has an associated probability listed immediately after the move name.
- Outcome number – a unique number identifying each outcome, or the number 0 representing a null outcome.
- Outcome name - a name for the given outcome, optional.
- Payoffs – a list of payoffs ordered from player 1 to the highest player represented in the game. Payoffs may be separated by white space or commas. Optional.

The file is divided into two sections – the header (or prologue) and the body. The header contains information about the file format and the game description. The body of the efg file contains the game information in the form of a list of nodes.

There are three types of nodes that may be found in the file body: player (decision) nodes, chance nodes, and terminal nodes. The first unquoted letter in the line defines the node type for the information in that line. For example, a first character of "c" indicates the node is a chance node.

The player (decision) node contains information about the node itself (for example, the node name and associated player), as well as information about the moves that may be made by the player at this node. The move information at each player node allows the file to be parsed top-down in a deterministic way. Since the node's expected children are defined implicitly by the moves within the move section, the subsequent rows can be correctly categorized as children of the node, or children of another node.

Figure 3. An annotated player node line.

Like the player node, the chance node type contains information about the chance node itself, as well as information about the moves that may be made from that node. The chance node also contains the probability information associated to each move. No player is involved at a chance node – only probability comes into play. Other than these two differences, the player and chance nodes are similar in structure.

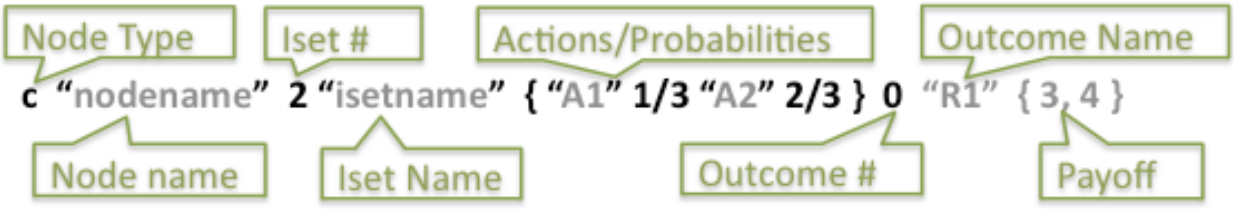


Figure 4. An annotated chance node.

The terminal node is a leaf node in the tree. Like the player and chance nodes, the terminal node may have a node name, outcome number, and payoff. No decisions are made at this node – it represents an end state only.

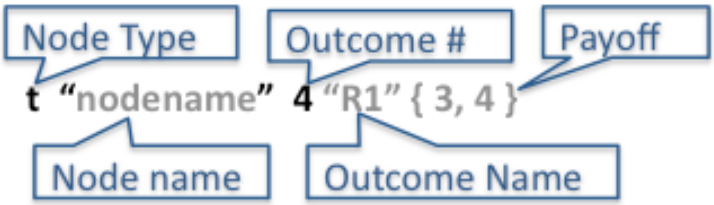


Figure 5. An annotated terminal node.

A sample game is depicted in Figure 6. The first node is a chance node with two children; the two children are found on lines 5 and 10. The rest of the nodes, which are children or grandchildren of the first two player nodes are found underneath the parent nodes. Note that the game tree corresponding to this file can be found in Figure 7.

```
1     EFG 2 R "Untitled Extensive Game" { "Player 1" "Player 2" }
2     ""
3
4     c "root" 1 "" { "Chance 1" 1/2 "Chance 2" 1/2 } 0
5     p "P2 - 1" 2 1 "" { "a" "b" } 8 "" { 1, 5 }
6     t "" 1 "" { 14, 15 }
7     p "P1 - 1" 1 1 "" { "C" "D" } 0
8     t "" 2 "" { 24, 10 }
9     t "" 3 "" { 21, 19 }
10    p "P2 - 2" 2 1 "" { "a" "b" } 9 "" { 2, 3 }
11    p "P1 - 2" 1 2 "" { "E" "F" } 0
12    t "" 4 "" { 2, 14 }
13    t "" 5 "" { 23, 10 }
14    p "P1 - 3" 1 2 "" { "E" "F" } 0
15    t "" 6 "" { 2, 18 }
16    t "" 7 "" { 4, 7 }
```

Figure 6. An example efg file.

nfg files

There are two nfg file formats that may be used to represent a strategic game [3]. The first type is the payoff version. In this type of file there is a header and a body. The header contains information about the game description and player names. The body of the file contains the payoff information.

```
1       NFG 1 R "Two person 2 x 2 game with unique mixed eq." { "Player 1" "Player 2" }
2       { 2 2 }
3
4       2 0 0 1 01 1 0
```

Figure 7. An example nfg payoff version file.

Each pair of numbers on line 4 in the file above is a pair of payoffs in the strategic game. The "counting off" of those pair to match them to the strategies can be done as described on the gambit project website:

> *In general, the ordering of contingencies is done in the same way that we count: incrementing the least-significant digit place in the number first, and then incrementing more significant digit places in the number as the lower ones "roll over." The only differences are that the counting starts with the digit 1, instead of 0, and that the "base" used for each digit is not 10, but instead is the number of strategies that player has in the game.* [3]

In the example the first pair of numbers on the payoffs line (2 0) represents the payoffs for player 1 and player 2 choosing their first strategy. The value 2 is player 1's payoff, and the value 0 is player 2's payoff. The second pair of numbers (0 1) represents the payoffs for player 1 playing his first strategy, and player 2 playing his second strategy. One can continue in this manner along the payoffs line to match each pair of payoffs to the combination of strategies that will produce each set.

The second type of nfg file is called the outcome version. In this file type the information is divided into several sections. The game description and player names and order can be found in the header section. Following this section, the strategies names (labels) are listed for each player. Next, the payoff combinations are listed; note that each payoff starts with a text field delimited by "", which corresponds to the outcome name. This outcome name is ignored during processing.

In the last section of the file the payoffs are mapped to the strategic game outcomes. Each number in the last line of the file refers to the payoffs listed previously and is known as an outcome index. The indices map the payoffs sets listed above to the position in the payoff matrix. The position of the number indicates the location in the payoff matrix where the payoff belongs. The files in Figures 7 and 8, although they look different, encode the same game and will map to one common XML output.

```
1     NFG 1 R "Two person 2 x 2 game with unique mixed eq." { "Player 1" "Player 2" }
2
3     { { "1" "2" }
4     { "1" "2" }
5     }
6     ""
7
8     {
9     { "" 2, 0 }
10    { "" 0, 1 }
11    { "" 0, 1 }
12    { "" 1, 0 }
13    }
14    1 2 3 4
```

Figure 8. An example nfg payoff version file.

LaTeX strategic game files for the bimatrixgame.sty macro

The bimatrixgame.sty macro can be used to format a strategic game in a visual way suitable for printing, sharing, and publication. The macro allows for shortcut text for a particular presentation request or task to be entered into the LaTeX file and transformed into the appropriate representation. The macro resolves the shortcut into it the full set of commands that LaTeX can interpret in order to generate the requested output. Details can be found in [7]. An introduction to LaTeX can be found in [8].

```
1     \documentclass{article}
2     \usepackage{bimatrixgame}
3     \usepackage[usenames]{color}
4     \begin{document}
5
6     \renewcommand{\bimatrixrowcolor}{Red}%
7     \renewcommand{\bimatrixcolumncolor}{Blue}%
8     \renewcommand{\bimatrixdiag}{0}%
9     \renewcommand{\bimatrixpairfont}{\small}%
10
11    \bimatrixgame{4mm}{3}{2}{P1}{P2}%
12    {{T}{M}{B}}%
13    {{l}{r}}%
14    {
15    \payoffpairs{1}{{$1$}{$-2$}}{{$-2$}{\fbox {$5$}}}
16    \payoffpairs{2}{{$3$}{$\frac{5}{4}$}}{{$-3$}{\fbox {$1$}}}
17    \payoffpairs{3}{{\fbox {$5$}}{\fbox {$6$}}}{{$\frac{7}{8}$}{\fbox {$3$}}}
18    }
19    \end{document}
```

Figure 9. An example .tex file formatted for the bimatrixgame macro.

Line 1 of the file indicates to the LaTeX processor that the document should be handled as an *article*.

Lines 2 and 3 indicate that macro packages should be referenced when processing the document text. The first package is the bimatrixgame.sty macro, the format of which we review in detail below. The second package is the color package which maps the names of defined colors to their hexadecimal equivalents [9, 10]. This package is external to the Gambit project and is not reviewed in this document.

The \begin command on line 4 indicates to the processor that the contents of the document follow this line. The \renewcommand directives found on lines 6-9 allow the user, if desired, to override defaults found in the macro.

The bimatrixgame command on text line 11 indicates that the inputs to the macro will start here. There are five inputs on this line, each contained within a pair of matching braces, "{}". The first input to this line is the height and width of the strategic game matrix payoff cell; in the example this size is 4mm. The second and third inputs correspond to the number of strategies of the row and column player, respectively. In the example above, the game is a 3x2 game. The row player has 3 strategies, and the column player has 2 strategies. Finally, the last two parameters represent the player names for the row and column player. The row player has name "P1" and the column player has name "P2."

Lines 12 and 13 define the names of the strategies for the row and column player. Following the convention of listing the row player first, the first row contains the labels for the row player's strategies: "T", "M", and "B." Each strategy label is contained within a set of matching braces. The next row contains the labels for the column player's strategies. These are "l" and "r" in the example.

The information starting on line 14 lists the player's payoffs, with some additional formatting information. The command \payoffpairs appears once for each one of the row player's strategies. Within this command can be found a matching payoff pair for each of the column player's two strategies, as follows.

\payoffpairs{1}{{1}{-2}}{{-2}{\fbox {5}}}

The text highlighted in orange above corresponds to the payoffs received by player P1 and player P2 in response to player P1 playing his first strategy ("T"), and player P2 player his first strategy ("l"). The payoff to player P1 is "1", and the payoff to player P2 is "-2." The text highlighted in green corresponds again to the payoffs received by player P1 and player P2. In this case, these payoffs are in response to player P1's first strategy ("T") and player P2's second strategy ("r"). The fbox command for player P2's payoff indicates that the payoff should be highlighted by a box around the value, which by convention shows visually that strategy "r" is a best response to player I's strategy of "T." The following two "\payoffpairs" lines can be interpreted in a similar way.

It is also possible for the game to be of the "single payoff" variety. In this case, the payoffs for both player I and II, given a set of strategies, is the same. The command used is "\singlepayoff" and only one payoff is given per strategy combination, as follows:

```
\singlepayoffs{1}{{$\frac{1}{4}$}{$1$}}
```

The information highlighted in orange corresponds to the payoff for both players playing their first strategy. The amount is a fractional amount, and is to be formatted with a horizontal bar between the numerator and denominator. The information highlighted in green is the payoff for player P1 playing his first strategy and player P2 playing his second strategy.

Additionally, note that the formatting markers {$ and $} are used for each payoff value. These formatting delimiters ensure that negative numbers are properly formatted. They are not required for positive number, as they have no effect on those numbers, but for consistency an ease of processing all numbers are contained within these indicators.

Finally, the "\end" command on line 19 indicates that the document is complete. When the bimatrix macro is applied and the output converted to dvi we get a display as shown in Figure 10.

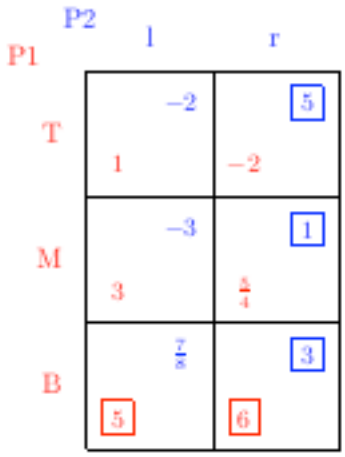


Figure 10. Example output display formatted by the bimatrixgame macro.

XML Design Approach

The XML structure for extensive and strategic games should be general and robust enough to include any and all features currently supported by the Gambit file formats such that the XML structure could potentially also be used for Gambit files in the future. To such an end the XML structure proposed in this document is informed by the current Gambit efg and nfg formats, in that all elements supported by these file formats should also be supported by the gte XML structure such that lossless data conversion can be achieved between the formats.

Additionally, the XML structure should separate display information from data so that different consumers of the data can focus on the data that is relevant to them. For example, if a user would like to compute equilibria for a particular strategic game, they will not need the display information.

Proposed display elements for the XML representation are not covered in this document, although the placeholder *display* tags will be included to indicate where this data will be stored in the future.

XML Approach: Overview

The XML specification outlined in this document builds on the existing gte XML specification. Differences between the specifications are briefly explored in Appendix A. In this section we focus on discussing the XML structure and how it supports communication of the game data.

Each XML document must have a root element that signals the beginning of the document's information. In this case we will choose *gte* as the root element. The gte root can have five possible children.

The first child is the game description. The game description element contains text describing the game. This element maps directly to the game description element within the efg and nfg file formats, and should be present for all documents, although the contents may be blank.

The next child of gte, the players element, explicitly defines the name and ordering of each player. This element is required in all gte XML output.

The third child of gte contains information regarding how the game should be displayed on a UI. How the XML for the display should be formatted will not be covered in this document, but will be addressed at a later time. However, in this document the display tags are shown as a placeholder in some contexts.

The remaining two child elements represent the type of game, extensive or strategic, and are an either/or choice for a particular document. One document cannot represent both an extensive and a strategic game.

As an example, the beginning formulation of an extensive game would look as follows, where "…" represents elements that will be filled in as we proceed exploring the structure.

```
<gte version="0.1">
    <gameDescription>Untitled Extensive Game</gameDescription>
    <players>
        <player playerId="1">Player  1</player>
        <player playerId="2">Player  2</player>
    </players>
    <display></display>
```

```
    <extensiveForm>
                …
    </extensiveForm>
</gte>
```

The beginning formulation of a strategic game is similar, except the extensiveForm tag is replaced by the strategicForm tag.

```
<gte version="0.1">
    <gameDescription> Untitled Extensive Game</gameDescription>
    <players>
        <player playerId="1">Player  1</player>
        <player playerId="2">Player  2</player>
    </players>
    <display></display>
    <strategicForm>
                …
    </strategicForm>
</gte>
```

Extensive games

Let's first explore the information that makes up the extensive game, starting from the smallest building block: outcome.

Outcome. A terminal node represents a node in the game tree that has no children. A simple terminal node with no payoff is represented by the outcome element:

```
<outcome />.
```

In fully defined games outcomes are likely to contain player payoffs. An payoff for Player 1 in a game can be represented as:

```
<payoff player="Player 1">5</payoff>.
```

Note that the element has a payoff content of 5, indicating that the payoff for Player 1 has the value 5. Payoffs may be child elements of the outcome element.

A terminal node with payoff for a two-player game with payoff 5 for Player 1 and payoff 7 for Player 2 would look as follows:

```
<outcome>
        <payoff player="Player 1">5</payoff>
        <payoff player="Player 2">7</payoff>
</outcome>
```

As mentioned earlier, each node in an efg file has an outcome id. If the outcome id is

zero, that is the null outcome. This does not need to be represented in the XML. However, if the outcome id is greater than 1 this will be represented in the XML as an attribute with name *outcomeId*:

```
<outcome outcomeId="1">
    <payoff player="Player 1">5</payoff>
    <payoff player="Player 2">7</payoff>
</outcome>
```

Note that the payoff elements can be repeated to communicate the payoffs for any and all number of players participating in an extensive game.

Nodes. Both a chance node and a player (decision) node are represented by the same element: node.

The attributes of the node define what type of data the node represents. Node attributes include the iset, outcome, and player. The attributes define the information at the node itself, as well as potentially information about its parent. For example, if a node has an associated probability, its parent node was a chance node, and its siblings are also chance nodes with associated probabilities. As another example, if a node contains either iset name or number, we know the node is part of an iset.

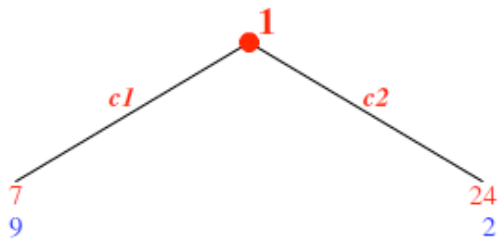


Figure 11. A simple game created with gte.

In the extensive game shown in Figure 11 the root node represents a player node with two possible moves. Each move results in different payoffs to the two players participating in the game. The Figure 11 scenario can be represented in an efg file as shown in Figure 12.

```
1       EFG 2 R "Two Choices" { "Player 1" "Player 2" }
2       ""
3
4       p "root" 1 1 "" { "c1" "c2" } 0
5       t "result1" 1 "" { 7, 9 }
6       t "result2" 2 "" { 24, 2 }
```

Figure 12. The efg file which captures the information of the extensive game shown in Figure 11.

The XML corresponding to the information in figure 12, omitting the gte and initial game description elements for brevity, is

```
<node iset="1" nodeName="root" player="Player 1">
    <outcome move="c1" nodeName="result1">
        <payoff player="1">7</payoff>
        <payoff player="2">9</payoff>
    </outcome>
    <outcome move="c2" nodeName="result2">
        <payoff player="1">24</payoff>
        <payoff player="2">2</payoff>
    </outcome>
</node>
```

Some observations about the above XML. The root node contains attributes to represent the player that is making the decision as well as the iset number that is present in the efg file. The outcome elements in the XML have a parent element of node. The move information on line 4 of the efg file is associated to the node or outcome that is a result of that move. That is, "c1" is the name of the first outcome, and "c2" is the name of the second outcome. Each node and outcome in the example has a nodeName.

A game similar to that in Figure 11 is shown in Figure 13. In this case the root node is a chance node, with the left outcome having a probability of 13/20 and the right outcome having a probability of 7/20. The payoffs once again are different based on the move of the root that is selected by chance.

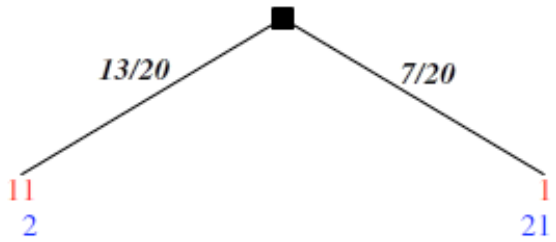


Figure 13. Simple extensive game with chance node, created with gte.

Figure 14 lists the efg file which represents the game in Figure 13.

```
1    EFG 2 R "Untitled Extensive Game" { "1" "2" }
2    ""
3
4    c "" 1 "" { "m1" 13/20 "m2" 7/20 } 0
5    t "" 2 "" { 11, 2 }
6    t "" 3 "" { 1, 21 }
```

Figure 14. The efg file represents the information of the extensive game shown in Figure 13.

The XML for the node data, omitting header elements for brevity, can be written as

```
<node iset="1">
    <outcome move="m1" prob="13/20" outcomeId="2">
```

```
            <payoff player="1">11</payoff>
            <payoff player="2">2</payoff>
        </outcome>
        <outcome move="m2" prob="7/20" outcomeId="3">
            <payoff player="1">1</payoff>
            <payoff player="2">21</payoff>
        </outcome>
    </node>
```

Note that the nodes in the Figure 14 efg file do not have node names. Correspondingly, there is no nodeName attribute in the XML. If an attribute value does not exist, or has a blank value, it is not be represented in the XML attributes. Additionally, note the presence of the *prob* attribute in the outcome tag. This attribute indicates that the outcome is a result of a chance move that occurred with the given probability value. For a game tree the move and probability are data associated to the edge of the graph, and the remaining attributes such as player, iset, etc., are data associated to the node itself. In an XML representation of the game tree it would be possible to have both a probability and a player at the same node if the node is a player decision node, but was a result of a chance move.

Payoffs may be associated to an internal node, and nodes may be nested to any depth – just as a game tree can have any number of levels.

The following diagram represents the game tree represented by the efg file in Figure 6.

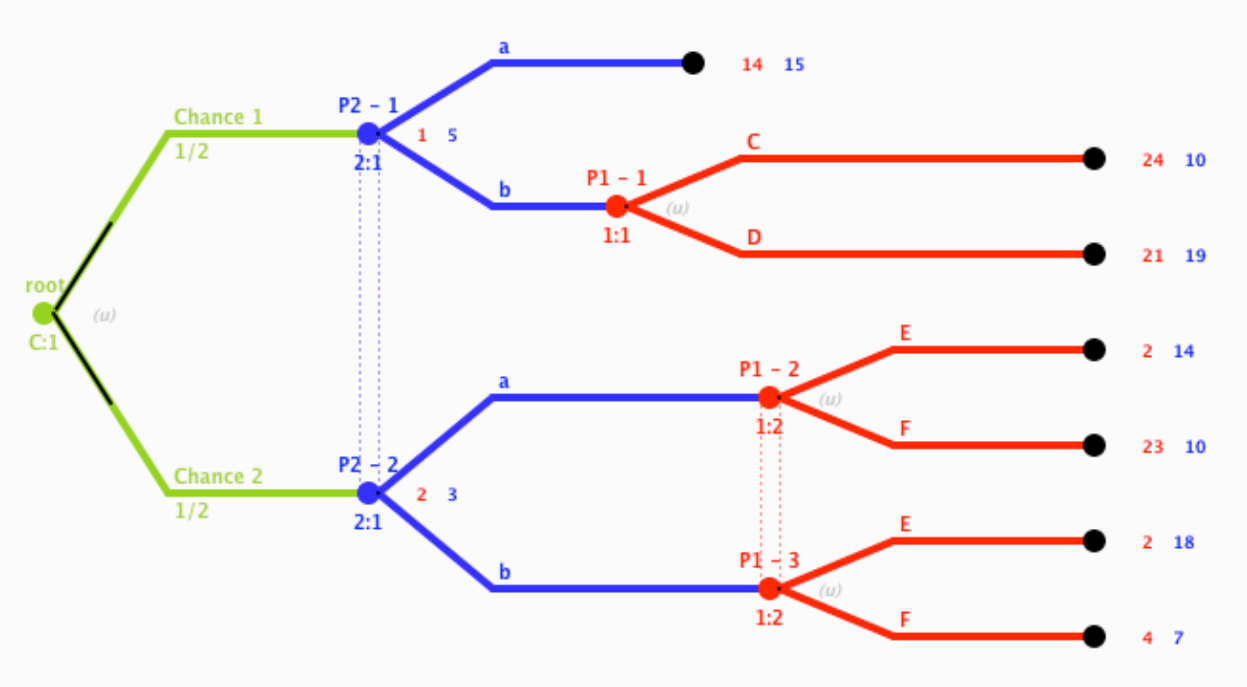


Figure 15 - extensive form game created in Gambit.

This game is represented in XML as

```
<gte version="0.1">
    <gameDescription>Sample Game</gameDescription>
    <players>
        <player playerId="1">Player  1</player>
        <player playerId="2">Player  2</player>
    </players>
    <display></display>
    <extensiveForm>
         <node nodeName="root">
             <node iset="1" move="Chance 1" nodeName="P2 - 1" player="Player 2"
prob="1/2">
                 <payoff player="Player 1">1</payoff>
                 <payoff player="Player 2">5</payoff>
                 <outcome move="a">
                     <payoff player="Player 1">14</payoff>
                     <payoff player="Player 2">15</payoff>
                 </outcome>
                 <node iset="2" move="b" nodeName="P1 - 1" player="Player 1">
                     <outcome move="C">
                     <payoff player="Player 1">24</payoff>
                     <payoff player="Player 2">10</payoff>
                     </outcome>
                     <outcome move="D">
                          <payoff player="Player 1">21</payoff>
                         <payoff player="Player 2">19</payoff>
                     </outcome>
                 </node>
             </node>
             <node iset="1" move="Chance 2" nodeName="P2 - 2" player="Player 2"
prob="1/2">
                 <payoff player="Player 1">2</payoff>
                 <payoff player="Player 2">3</payoff>
                 <node iset="3" move="a" nodeName="P1 - 2" player="Player 1">
                     <outcome move="E">
                         <payoff player="Player 1">2</payoff>
                         <payoff player="Player 2">14</payoff>
                     </outcome>
                     <outcome move="F">
                          <payoff player="Player 1">23</payoff>
                          <payoff player="Player 2">10</payoff>
                     </outcome>
                 </node>
                 <node iset="3" move="b" nodeName="P1 - 3" player="Player 1">
                     <outcome move="E">
                        <payoff player="Player 1">2</payoff>
                        <payoff player="Player 2">18</payoff>
```

```
                    </outcome>
                    <outcome move="F">
                      <payoff player="Player 1">4</payoff>
                      <payoff player="Player 2">7</payoff>
                    </outcome>
                  </node>
                </node>
              </node>
            </extensiveForm>
</gte>
```

Strategic Games

A strategic game contains strategy information like an extensive game. However, since it does not contain nested or tree information it is more straightforward to represent and map this type of game in XML. The main concepts communicated in the strategic game are the strategies and the payoffs. See Figure 2 and Figure 10 for examples of strategic games.

Starting with the pieces that make up the strategic game XML, the strategy element contains the labels for a particular player's strategies. The strategies for a two-player game where each player has two strategies may be represented by

```
<strategy player="Player 1">{ "1" "2" }</strategy>
<strategy player="Player 2">{ "1" "2" }</strategy>
```

We can see the first set of strategy labels belong to Player 1, as listed in the attributes section of the strategy tag. This is consistent with the way player data is represented in extensive games. The content of the strategy tags are quoted identifiers which represent the player's strategies in order. In this way we can represent strategies for any number of players. In the first line above the, we see that Player 1 has two strategies: "1" and "2". Player 2 also has two strategies, with the same labels as Player 1's strategies. If there were more players their strategies would continue the list.

A player's payoff is listed within the payoffs tag. The payoffs follow the same ordering convention as the nfg files. Below is an example of player payoffs – each payoff is delimited by whitespace. Rows are separated by the newline "\n" character. The *payoffs* tag is used for the strategic game, compared to the *payoff* tag used for the extensive game. The two elements are not interchangeable.

```
<payoffs player="Player 1"> 2 3
 5 7
</payoffs>
<payoffs player="Player 2"> 3 3
6 9
</payoffs>
```

		1			2		
c	A	2	1	3	3	1	3
	B	0	2	1	0	2	0
d	A	5	5	6	7	4	9
	B	0	7	0	1	-2	1
e	A	1	2	0	0	0	3
	B	1	-1	1	7	5	0

Figure 16. A three person strategic game viewed in Gambit.

A complete three person strategic game, depicted in Figure 16 as displayed by Gambit, can be represented in XML as

```
<gte version="0.1">
    <gameDescription>My untitled game</gameDescription>
    <players>
        <player playerId="1">Player  1</player>
        <player playerId="2">Player  2</player>
        <player playerId="3">Player  3</player>
    </players>
    <display></display>
    <strategicForm size="{ 3 2 2 }">
        <strategy player="Player 1">{ "c" "d" "e" }</strategy>
        <strategy player="Player 2">{ "1" "2" }</strategy>
        <strategy player="Player 3">{ "A" "B" }</strategy>
        <payoffs player="Player 1"> 2 5 1
        3 7 0
        0 0 1
        0 1 7
        </payoffs>
        <payoffs player="Player 2"> 3 6 0
        3 9 3
        1 0 1
        0 1 0
        </payoffs>
        <payoffs player="Player 3"> 1 5 2
         1 4 0
         2 7 -1
         2 -2 5
        </payoffs>
    </strategicForm>
</gte>
```

For a two player game the payoff for a player can be represented by an M x N matrix. M is the number of rows, or strategies, for the first player. N is the number of columns, or strategies, for the second player. If there are more than 2 players then there are

multiple M x N matrices representing the payoffs for the strategies of the additional players. For a three player game, as above, where the third player has two strategies, we expect the M x N matrix to be repeated 2 times; once for each of the third player's strategy. This approach can be extrapolated to games for any number of players.

Conversion Logic Implementation

EFGToXML

The EFGToXML java class converts files in the Gambit efg format to the gte extensive form XML format. The class reads the input file, extracts the information from the efg file format, and transforms the information into the specified extensive form XML format.

As discussed earlier, in an extensive form game each player or chance node has a list of possible moves that may be made from that node. The list of moves, defined at the node level and listed on the same line as the node information in the efg file, allows the program to determine the number of children for that node at the point when that node's information is read.

The algorithm for reading and interpreting the efg file, at a high level, works as follows. Any data that are not part of the extensive game tree itself (for example, the game description or player names and order) are added to the XML document in a straightforward manner. For the game tree a stack keeps track of the parent nodes to which additional children node must be added. The stack is created and the root added to the stack. When each node is added to the stack a temporary variable that records the expected number of children for that node is associated to the node. The expected number of children for a node is calculated by counting the number of moves possible from that node as listed in the efg file.

Each line of text in the file represents a node in the extensive game tree. As the program processes each line of text in the file, the logic determines the correct parent is for that node.

If the node is a terminal node, it's parent is the top node on the stack and no further searching is required. The top node on the stack remains on the stack as children are added. Only the node at the top of the stack may have any children appended.

If the node is a non-terminal node the number of children of the potential parent (the node at the top of the stack) is calculated. If the expected number of children matches the actual number of children for the top node of the stack, then the top node is popped off the stack. The new top node undergoes a similar comparison of expected number of children to actual number of children; this goes on until the top node of the stack still has capacity to have more children added. When the correct parent is found for the current node it is attached and the next node is processed.

When all of the nodes in the efg file have been processed, the remaining parent nodes on the stack are popped and the expected children variable is cleared and removed from the node.

The gte tool requires all moves to have names. Therefore, if a move in the efg file is blank, it will be given a default name when generating the XML. Default moves start with _ (underscore) that is followed by a unique number that will follow in sequence any previous default move numbers.

Constructor and Methods

(Constructor)	*EFGToXML(String fn)* Creates an instance of the *EFGToXML* class and takes the name of the file to be converted as an input.
void	*setFileSuffix(String suffix)* This method allows the user to set an alternate suffix and extension for the output file. For example, the default suffix is ".xml", but an entry of "_Jan1.xml" will append the entire suffix to the input file name, transforming a file named "example.efg" to "example_Jan1.xml."
void	*setTestMode(boolean tm)* If the test mode is set to true the DTD definition element will be created as part of the XML.
void	*setDTD(String d)* Sets the filename of the DTD.
void	*convertEFGToXML()* Initiates the conversion process, which is described above in more detail. The output file is written to the directory where the program is invoked.

XMLToEFG

The XMLToEFG class converts a file from the gte extensive game XML format to an efg file appropriate for use with Gambit.

The header level XML data, such as player names, and game description, are parsed in to private members of the class and written to the efg file in a straightforward manner. The blank quotation marks that are a default attribute of the sample files is also written to line 2 of the output.

The XML game tree is then traversed. The traversal logic creates a line in the efg file to correspond to each node in the XML. The node element contains all information for the node in the efg file with the exception of the moves and probabilities. The move name, and probabilities if relevant, are read from the children of the node and written to the file for the appropriate parent node. In other words, the efg file lists the move information (information related to the edge leading away from the node) at the node itself, whereas the XML format associates that edge information to the child (result) of the move.

In an efg file payoff values may be separated by either whitespace or commas. When an XML file is used to generate an efg file the payoffs will be separated by commas.

Constructor and Methods

(Constructor)	*XMLToEFG(String fn)* Creates an instance of the *XMLToEFG* class taking the name of the file to be converted as an input.
void	*setFileSuffix(String suffix)* Similar to setFileSuffix of EFGToXML. By default the *convertXMLToEFG()* method will take a *.xml file and create a *.efg file. This method allows the user to set an alternate suffix and extension for the output file.
void	*convertXMLToEFG()* Initiates the conversion process. The output file is written to the directory where the class is invoked.

NFGToXML

The NFGToXML class takes a file in Gambit nfg format and transforms it into an XML file meeting the gte strategic form XML format.

The conversion of the nfg file format to XML is fairly straightforward and for the most part minimal logic is required to map the elements from one format to another.

For the payoff format processing the payoff pairs as listed in the file must be separated and organized into the appropriate matrix structure for output in the XML format. The format requirements are discussed in the section regarding the XML proposal.

For the outcome format, the payoff pairs must be organized in the correct order, since the outcomes may be listed out of order of the actual mapping to the strategies - see the section on nfg file formats earlier in this document for information on the outcome indices. Once the pairs are ordered correctly, they are organized into the appropriate matrix structure for XML output.

Constructor and Methods

(Constructor)	*NFGToXML(String fn)* Creates an instance of the *NFGToXML* class using the name of the file to be converted as an input.
void	*setFileSuffix(String suffix)* Similar to setFileSuffix of strategicFileToXML. By default the conversion takes a *.nfg and creates a *.xml file. This method allows the user to set an alternate suffix and extension for the output file.
void	*setTestMode(boolean tm)* When test mode is set to true the DTD definition element will be created as part of the XML.
void	*setDTD(String d)* Sets the name of the file that represents the DTD for the XML.
void	*convertNFGToXML()* Initiates the conversion process.

XMLToNFG

The XMLToNFG class converts an XML representation of a strategic game into an nfg representation of the game. The output file is in one of the two Gambit nfg formats: *outcome* or *payoff*. In either case, the blank quotation marks on line 2 that are a characteristic of the sample files is written to file for consistency.

The mapping of the XML elements to the corresponding elements in the nfg file formats is straightforward. The only data manipulation required is the formatting of the payoffs. The method for formatting is the complement of the method used in the nfg to xml transformation. The outcome format file always lists the outcome indices in order.

Constructor and Methods

(Constructor)	*XMLToNFG(String fn)* Creates an instance of the *XMLToNFG* class and takes as input name of the file to be converted.
void	*setFileSuffix(String suffix)* Similar to setFileSuffix of strategicFileToXML. By default the <> method will take a *.xml file and create a *.nfg file. This method allows the user to set an alternate suffix and extension for the output file. In this case, a user could generate both the output and payoff versions of the same input, and differentiate them with a different suffix.
void	*setNFGFormat(String format)* Set the class to create either an output file in either outcome or payoff format. If no value is set through this method the default is outcome format.
void	*convertXMLToNFG()* Initiates the conversion process. The output file is written to the directory where the class is invoked.

StrategicFileToXML

A strategic game matrix file is a text file containing the payoff matrices for a particular strategic game. The StrategicFileToXML java class converts these matrix files into the gte strategic form xml representation.

The source file may contain one or two M x N matrices. For single payoff game only one matrix is included. For a game where each player has different payoffs, two matrices are included. The two matrices must be of the same dimensions. The first matrix contains the payoffs for player 1, the second matrix the payoffs for player 2.

Figure 17. Example of matrix flat files

The transformation of the data from the matrix representation to the XML representation is straightforward. Where required, some parameters are set to the default values.

Constructor and Methods

(Constructor)	*StrategicFileToXML(String fn)* Creates an instance of the StrategicFileToXML class taking the name of the file to be converted as an input.
void	*setFileSuffix(String suffix)* Provides an option to change the default text at the end of the filename, including the extension. For example given an input file of matrix22.xml, an input to this method "_20110801.xml" will generate the file matrix22_20110801.xml as an output.
void	*setTestMode(boolean tm)* If test mode is set to true the dtd definition element will be created as part of the XML.
void	*void setDTD(String d)* Sets the filename for the XML's DTD.
void	convertFlatFileToXML() The method that orchestrates and completes the conversion. Takes the text file used in the constructor and transforms the information from text to XML. Creates the output file in the same directory as the program is run.

XMLToLaTeX

The XMLToLaTeX class takes a file of gte strategic form XML format and creates a LaTeX (.tex) file that represents the game. The initial file could be the output of the strategicFileToXML class, or it could be generated by an NFGToXML conversion. The contents of the .tex file conform to the requirements of the bimatrixgame.sty LaTeX macro. This conversion program creates only the tex file. To create the .dvi file which can be used for viewing or printing, the LaTeX program is applied separately to resolve the macro and create the formatted document.

Within the XML display element there are several options that define non-default behavior of the macro output. Although the display element is not discussed in detail in this document, the options available in this element do affect the output of this particular conversion, and as such are discussed here. The names presented in this document are names for the options from the conversion program. The actual element names for the XML are to be determined at a future date. Within the logic "stand-in" names for the XML elements have been used; these will be updated at a later time when the naming conventions are finalized. The concepts are expected to remain consistent.

rowColor – The color to represent the row player's payoff. The colors can be, for example, Red, Blue, etc. The macro uses a package that maps hexadecimal color values to color names, so only the color names specified in the documentation [9] and [10] are supported.

colColor – The color to represent the column player's payoff. The allowed colors are the same as for the rowColor.

cellSize – a measurement in mm, such as "4mm", that represents the lengths of the width and height of the payoff cell in the strategy matrix.

diagSize - size of the diagonal bar that separates the player names on the top left of the output.

pairFont – There are three allowed sizes: small, normalsize, and large. This is the size of the font when the payoffs are in pairs.

singleFont – Similar to above, there are three allowed sizes: small, normalsize, and large. This is the size of the payoff for "single Payoff" mode.

SinglePayoff – A true/false setting. If singlePayoff is true then payoffs are the same for player 1 & player 2 and displayed as one value. If single payoff is false, then a possibly distinct value for each player is displayed in the strategy matrix.

PrettyFraction – A true/false setting. A true value indicates the fraction should be shown with a horizontal bar instead of diagonal bar separating numerator and denominator

ShowBestResponse – A true/false setting. If the option is true tthe best response is calculated and highlighted with a box around the value.

Constructor and Methods

(Constructor)	*XMLToLaTeX(String fn)* Creates an instance of the XMLToLaTeX class taking the name of the file to be converted as an input.
void	*setFileSuffix(String suffix)* Similar to setFileSuffix of strategicFileToXML. By default the *convertXMLToLaTeX()* method will take a *.xml file and create a *.tex file. This method allows the user to set an alternate suffix and extension for the output file.
void	*convertXMLToLaTeX()* Initiates the conversion process of the XML file to the .tex file. Creates the output file in the same directory as the program is run.

Testing

The testing approach for the project consisted of a combination of manual and automated testing. Files were validated by manual inspection with automated support for the bulk creation of converted files and DTD validation. Files were also validated by using the Gambit and gte programs to verify the files yielded the expected game display. Additionally, JUnit tests were created in order to be able to quickly validate small changes done after the initial development, and to assist any future developers who need to validate the deployment of the code to their local system.

EFGToXML and XMLToEFG Testing approach

Within the Gambit distribution there are approximately 90 sample efg files, consisting of various types of extensive games, with different payoff structures, number of players, tree morphology, etc. The approach in testing the efg to XML and XML to efg conversion was to use these files as the basis of the testing, with supplemental files to ensure coverage of all test conditions. There were two main test approaches used.

The first test generated a "round trip conversion" of the files. First the efg sample files were converted to the XML format. The resulting XML files were both manually inspected for conformity to the XML specification as well as validated against the DTD. After validating the XML, the converted XML files were converted back to efg files. Then for each original efg file, there was a corresponding XML file, as well as an efg file created by the conversion back from XML to efg.

The efg files created by the round trip conversion were validated in two ways. First, a file difference utility was used to compare the equivalent files and confirm that they matched in all areas were an exact match was expected. Note that the efg files are expected to match exactly with the exception of the iset number, which may be different

between files; this difference is due to a difference in iset processing by Gambit and gte which may be updated in gte in the future. Secondly, the efg files generated from the XML were opened in the Gambit program to confirm the game trees of the original and new files matched.

The second test, an integration test, consisted of opening the XML files created by the conversion in the gte program. Any issues with opening the files in the program were evaluated to determine the source of the issue and sent to the appropriate party for resolution (as changes to the gte program were managed by another student).

The GeneralTest and the EFGToXMLTest classes were used to support automation of certain aspects of the tests. These two classes are described in more detail later in this document.

NFGToXML and XMLToNFG Testing approach

The Gambit distribution also contains approximately 50 sample nfg files. The strategic games in the nfg files contains differing number of strategies, players, and payoff structures. The testing approach for the nfg and XML conversions was similar to the test approach for the efg and XML conversions. This set of delivered files, along with supplemental files designed to cover any gaps in test conditions, were used as the basis for the test data.

The main testing approach for the nfg files was "round trip conversion." The nfg sample files were converted to the XML format. That XML files were validated to ensure all data from the nfg file was correctly represented. The XML output files were also validated against the gte.DTD, to ensure the format of the XML matched the specification. When validation was successful the converted XML files were converted back to nfg files. For each original nfg file the testing produced an XML file as well as two other nfg files. Two return conversion nfg files were generated as there are two nfg file types: outcome format and payoff format.

The nfg files created by the round trip conversion were validated in two ways. The first validation used a file difference utility to compare the equivalent files and confirm that they matched in all areas were an exact match was required. If the original file was a payoff format nfg file, it was compared to the newly created nfg payoff format file created. If the original file was an outcome format file, it was compared to the newly created nfg outcome format file. In this case, it is possible that the outcome index numbers might be different if the original file had non-sequential index numbers. The second validation for this test cycle consisted of opening the nfg files in the Gambit program and confirming that the strategic game matched the strategic game information from the original nfg test file.

Further testing with the gte tool can perhaps be initiated in the future when the gte tool accepts nfg format files.

StrategicFileToXML and XMLToLaTeX test approach

The testing of this set of programs consisted of a straightforward validation of the input and output of each of the programs based on test files created for this test. Additionally, selected strategic form XML files created as an output of the nfg set of tests were also used to create sample LaTeX files.

First a set of flat file matrix input files were run through the StrategicFileToXML program. The XML was validated by inspection and comparison to the original file.

Next, the XML files created as part of the first step were transformed into LaTeX files (.tex extension). These files were manually transformed to LaTeX documents by use of the latex command line utility, which produces a .dvi file. Once converted to .dvi the LaTeX files could be viewed using a dvi viewer. The results of the transformation were inspected and validated manually against the original input found in the test files.

Additionally, selected nfg files were transformed to LaTeX format by application of the XMLToLaTeX program. Then the files were processed by the latex command and manually inspected using a dvi viewer, and the output compared to the original XML input.

The automation of some of the test steps was accomplished by using the GeneralTest class which processes files in bulk from a specified directory.

Test Class Implementation

There are four test classes used to support testing.

GeneralTest

The GeneralTest class contains public methods to automate the bulk transformation of files in specified directories. The private member variables efgFilePath, nfgFilePath, and latexFilePath need to be updated by the individual doing the testing. They should be updated to point to the directories where the test files for each transformation type are stored on the tester's system.

(Constructor)	*GeneralTest* Creates an instance of the *GeneralTest* class.
void	*allEFGToXML()* Converts all the files in the efgFilePath that are of extension .efg into xml files, ignoring all other files. Prints a list of the files it is converting to standard output.
void	*allNFGToXML()* Converts all the files in the nfgFilePath that are of extension .nfg into xml files, ignoring all other files. Prints a list of the files it is converting to standard output.

void	*allStrategicFilesToXML()* Converts all the files in the latexFilePath that are of extension .txt into xml files, ignoring all other files. Prints a list of the files it is converting to standard output.
void	*allXMLToEFG()* Converts all the files in the efgFilePath that are of extension .xml into .efg files, ignoring all other files. Prints a list of the files it is converting to standard output.
void	*allXMLToNFG()* Converts all the files in the nfgFilePath that are of extension .xml into .nfg files, ignoring all other files. Prints a list of the files it is converting to standard output.
void	*allXMLToLaTeX()* Converts all the files in the latexFilePath that are of extension .xml into .tex files, ignoring all other files. Prints a list of the files it is converting to standard output.
void	*flatToLaTeX()* Calls allStrategicFilesToXML and allXMLToLaTeX in sequence, thereby converting any .txt matrix files into .tex files.
void	*roundTripEFG()* Calls allEFGToXML and allXMLToEFG in sequence, thereby converting any .efg files into XML files, and back to efg files. If the original filename for the efg file is original.efg, then the new file will have the name original_efg.efg to distinguish between the original and new files.
void	*roundTripNFG()* Calls allNFGToXML and allXMLToNFG in sequence, thereby converting any .nfg files into XML files, and back to nfg files. If the original filename for the nfg file is original.nfg, then there will be two new files with names original_pay.nfg and original_out.nfg, where the first file stores in the information in nfg payoff format, and the second file stores in the information in nfg outcome format.

EFGToXMLTest

The EFGToXMLTest class is a JUnit test and requires the JUnit 4.8.2 jar to run. It contains two public methods for conversion testing. In the second method of this file there appears a specific path to the location where the extensive form XML files to be validated are located. This path should be updated to the correct path for the individual tester's system.

To validate the XML against the DTD for the files in the specified folder, uncomment second Test rule; otherwise this test will not run as part of the JUnit test, which is the expected default behavior. The file path needs to be updated to the file path of the XML files on the tester's system. A copy of gte.dtd must be present in the same folder with the files to call the method successfully.

Methods

void	*testValidateEFGToXML()* Validates that a sample efg file created internally by the test method is converted to the correct XML format. The output is compared to a predefined output XML string. All files are cleaned up by the method before exiting. No input files are required from the file system for this test.
void	*testValidateEFGDTD()* This class will validate any XML files in the defined folder, which were created with reference to the DTD, against the DTD definition. If any elements do not match the DTD an error will be printed to standard output. The reference to the DTD in the XML output is created by the EFGToXML class when testMode option is set to true, and the DTD name supplied to the class as well. By default this method is commented out as a JUnit test, but it can be reactivated by removing the comment markers around the @Test directive.

NFGToXMLTest

Like the EFGToXMLTest class the NFGToXMLTest class is a JUnit test and requires the JUnit 4.8.2 jar to run. It contains two public methods for conversion testing. In the second method of this file there appears a specific path to the location where the strategic form XML files to be validated are located.

To validate the XML against the DTD for the files in the specified folder, uncomment second Test rule; otherwise this test will not run as part of the JUnit test, which is the expected default behavior. The file path needs to be updated to the file path of the XML files on the tester's system. A copy of gte.dtd must be present in the same folder with the files to call the method successfully.

Methods

void	*testValidateNFGToXML()* Validates that a sample nfg file created internally by the method is converted to the correct XML format. The XML is compared to a predefined output XML string. All files are cleaned up by the method before exiting, and no input files are required from the file system for this test.
void	*testValidateNFGDTD()* This class will validate any XML files in the defined folder, which were created with reference to the DTD, against the DTD definition. If any elements do not match the DTD an error will be printed to standard output. The reference to the DTD in the XML output is created by the NFGToXML class when testMode option is set to true, and the DTD name supplied to the class as well. By default this method is commented out as a JUnit test, but it can be reactivated by removing the comment markers around the @Test directive.

flatFileToLaTeXTest

The flatFileToLaTeXTest is a JUnit test and requires the JUnit 4.8.2 jar to run. It contains two public JUnit tests. This class does not contain DTD validation but may be updated in the future when the display element structure is finalized.

Methods

void	*testPayoffPairXML()* Validates that an XML file created from a matrix flat file with two matrices is correctly converted to the strategic XML format. This method does not have any file system dependencies.
void	*testSinglePayoffXML()* Validates that an XML file created from a matrix flat file with one matrix (single payoff) is correctly converted to the strategic XML format. This method does not have any file system dependencies.

References

[1] von Stengel, B. (2001), *Game Theory Basics*. Department of Mathematics, London School of Economics.

[2] Gambit Project (2011), *Game Representation Formats.* Accessed online at http://www.gambit-project.org/doc/formats.html#the-extensive-game-efg-file-format on July 25, 2011

[3] Gambit Project (2011), *Gambit: Software Tools for Game Theory.* Accessed online at http://www.gambit-project.org/doc/index.html on July 25, 2011

[4] *XML*. Accessed online at http://en.wikipedia.org/wiki/XML on July 25, 2011.

[5] *XML Basics*. Accessed online at http://www.xmlfiles.com/xml/ on July 26, 2011.

[6] XML DTD. Accessed online at http://www.xmlfiles.com/dtd/ on July 26, 2011.

[7] Bimatrixgame macro documentation. Accessed online at https://github.com/stengel/standalone/blob/master/example.pdf on August 12, 2011.

[8] Oetiker, Tobias (2011). The Not So Short Introduction to LaTeX 2e. Accessed online at http://mirror.math.ku.edu/tex-archive/info/lshort/english/lshort.pdf on June 1, 2011.

[9] LaTeX color package. Accessed online at http://people.oregonstate.edu/~peterseb/tex/samples/color-package.html on August 23, 2011.

[10] LaTeX color package. Accessed online at http://people.oregonstate.edu/~peterseb/tex/samples/docs/color-package-demo.pdf on August 23, 2011.

Appendix A: Additional notes on the XML design

There are a few straightforward differences in the proposed structure when compared to the XML structure that existed prior to July 2011. One difference is the header information included in the new proposal. Since the new game structure will support both extensive and strategic game formats, as well as additional information, such as display data and game description, it was necessary to create a new set of tags to capture this data, and to move the *extensiveForm* tag which was previously the root, to be a child element of the new root element, *gte*.

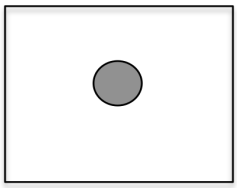
In the previous XML definition a terminal node with no payoffs was represented by the <node> element. In order to keep consistency in representation of a terminal node, regardless of payoff, the new XML proposal keeps the tag for a terminal node consistent as <outcome>.

Additionally, several new attributes for the node element are introduced in the new structure: *isetname, nodeName, outcomeId,* and *outcomeName*. The corresponding tags, where relevant, are also introduced for the outcome element: *nodeName, outcomeId,* and *outcomeName*.

Finally, the existing structure is updated to support strategic games. Strategic game information is contained within the *strategicForm* element when the XML file represents a strategic game.

The following two examples highlight the key structural differences between the new proposed XML and the existing XML.

Extensive Form, One Node no payoffs

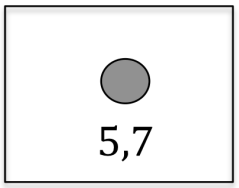


Notes:

- Degenerate case/starting scenario
- Note that in XML proposal outcome is used to represent any terminal node for consistency in representing the data

Current gte XML	<extensiveForm> <node/> </extensiveForm>
EFG File	EFG 2 R "Untitled Extensive Game" { "Player 1" "Player 2" } "" t "" 0
Proposed gte/Gambit format	<gte version="0.1"> <gameDescription> Untitled Extensive Game</gameDescription> <players> <player playerId="1">Player 1</player> <player playerId="2">Player 2</player> </players> <display></display> <extensiveForm> <outcome outcomeId="0"/> </extensiveForm> </gte>

Extensive Form - One Node with payoff



Notes:

- The value for GameDescription is optional, but the element is required for consistency in reading the XML

Current gte XML	<extensiveForm> <outcome> <payoff player="Player 1" value="5"/> <payoff player="Player 2" value="7"/> </outcome> </extensiveForm>
EFG File	EFG 2 R "Untitled Extensive Game" { "Player 1" "Player 2" } "" t "" 1 "" { 5, 7 }
Proposed gte/Gambit format	<gte version="0.1"> <gameDescription> Untitled Extensive Game</gameDescription> <players> <player playerId="1">Player 1</player> <player playerId="2">Player 2</player> </players> <display></display> <extensiveForm> <outcome outcomeId="1"> <payoff player="Player 1">5</payoff> <payoff player="Player 2">7</payoff> </outcome> </extensiveForm> </gte>

Appendix B: DTD

A Document Type Definition, or DTD, is a simple way to define the elements and attributes that make up a correctly formatted XML document for a specification [6]. Although other more powerful ways of formalizing XML documents exist, such as using an XML schema, in this document we will limit ourselves to defining our XML proposal in terms of a DTD as it is the simplest description.

The first portion of the DTD specifies the type of document (root) that is being defined.

The second portion of the DTD specification begins with a series of <!ELEMENT> tags that describe the XML elements which may be found in the document. The allowed content for the element, including children elements, is also defined. For example, when we read *<!ELEMENT gameDescription (#PCDATA)>* we know that the gameDescription element contains only content and no other elements or markup. Reading *<!ELEMENT extensiveForm (node|outcome)>* tells us that the extensiveForm element may have 1 child, either a node or an outcome. That is, its child is either the root node of a game tree with more than one node, or it is a simple tree with only one terminal node, <outcome>.

In the DTD the shorthand of using *, +, and ? to designate repeating elements is followed. The * symbol designates that a data element is repeated 0 or more times, the + symbol designates that a data element is repeated 1 or more times. The ? symbol indicates that a data element is present either 0 or 1 times. Then reading *<!ELEMENT node (node|outcome|payoff)*>* tells us that the node element may have zero or more children, and that those children may be either another node, an outcome node, a payoff, or some combination of all of those elements.

The final section of the DTD, which contains elements beginning with <!ATTLIST>, describes which attributes may be included within a given tag. The specification also defines whether that attribute is required (#REQUIRED) or optional (#IMPLIED). For example, the statement *<!ATTLIST strategicForm size CDATA #REQUIRED>* means that the *strategicForm* tag must always contain an attribute that lists the size of the game. Neither the DTD or XML specification make any claims about the order of the attributes.

```
<!ELEMENT gameDescription (#PCDATA)>
<!ELEMENT players (player)+>
<!ELEMENT player (#PCDATA)>
<!ELEMENT extensiveForm (node|outcome)>
<!ELEMENT node (node|outcome|payoff)*>
<!ELEMENT outcome (payoff*)>
<!ELEMENT payoff (#PCDATA)>
<!ELEMENT strategicForm (strategy*, payoffs+)>
<!ELEMENT strategy (#PCDATA)>
```

```
<!ELEMENT payoffs (#PCDATA)>

<!ATTLIST gte version CDATA #IMPLIED>
<!ATTLIST player playerId CDATA #REQUIRED>
<!ATTLIST node prob CDATA #IMPLIED>
<!ATTLIST node move CDATA #IMPLIED>
<!ATTLIST node player CDATA #IMPLIED>
<!ATTLIST node iset CDATA #IMPLIED>
<!ATTLIST node nodeName CDATA #IMPLIED>
<!ATTLIST node isetName CDATA #IMPLIED>
<!ATTLIST node outcomeId CDATA #IMPLIED>
<!ATTLIST node outcomeName CDATA #IMPLIED>
<!ATTLIST outcome move CDATA #IMPLIED>
<!ATTLIST outcome prob CDATA #IMPLIED>
<!ATTLIST outcome nodeName CDATA #IMPLIED>
<!ATTLIST outcome outcomeId CDATA #IMPLIED>
<!ATTLIST outcome outcomeName CDATA #IMPLIED>
<!ATTLIST payoff player CDATA #REQUIRED>
<!ATTLIST payoffs player CDATA #REQUIRED>
<!ATTLIST strategy player CDATA #IMPLIED>
<!ATTLIST strategicForm size CDATA #REQUIRED>
```

Appendix C: Standalone mode

The README_CONVERSION.txt document, which describes how to download and build the java conversion classes, as well as how to use them as conversion utilities, is reproduced below.

(1) Download the contents of the git repository kbletzer/standalone to a folder, following the clone github procedures. Make a note of the path to the standalone directory. The standalone directory is created in the location where the following git clone command is run. This folder should contain folders named test and lse, along with other files.

> git clone git@github.com:kbletzer/standalone.git

(2a) Within the script buildConversion.sh replace the placeholder text <path to standalone folder> with the actual path to the standalone folder noted in step 1.

(2b) Run the script buildConversion.sh in the standalone folder. The buildConversion.sh script will build all the java classes that are part of the conversion work. Note that the script does not assume any edits to the classpath variable - the path to the files is explicitly included in the build commands.

> sh buildConversion.sh

(3) Each of the six conversion classes takes a file as an input when run in "standalone" mode. The classes are

- EFGToXML - accepts a .efg format file and converts it to an extensive game format XML file
- NFGToXML - accepts a .nfg format file and converts it to an strategic game format XML file
- StrategicFileToXML - accepts a flat file with one or two matrices converts it to a strategic game format XML file
- XMLToEFG - accepts an extensive game form XML file and converts it to an .efg file
- XMLToNFG - accepts a strategic game XML file and converts it to an .nfg file
- XMLToLaTeX - accepts a strategic game XML file and converts it to a LaTeX file following the required format for the bimatrixgame.sty macro

From a directory containing the input files run the conversion program(s) as follows:

> java -cp <path to standalone folder> lse.standalone.EFGToXML e01.efg

The class name EFGToXML above can be replaced by any of the other conversion class names. The file e01.efg can be any file with the file format expected by the conversion program, as noted above. The <path to standalone folder> is the path noted

in step 1. If the path to the standalone folder is added to the java classpath, the -cp <path to standalone folder> option can be omitted.

The default output is a file with the same name as the input file and the extension updated to correspond with the output type. For the example above the output file will be e01.xml. Note that there is no error checking and each input file is assumed to meet the specifications for its file type.

(4) To change the extension of the target file and avoid overwriting any of the original files when making multiple conversions, run the command with an additional parameter for the extension as follows:

> java -cp <path to standalone folder> lse.standalone.EFGToXML e01.efg *_new.xml*

"_new.xml" will be appended to the filename instead of the default .xml extension. This feature is the same for all six conversion programs, although the extensions will be different for different programs.

Appendix D: In progress issues

Current issues/potential future updates from Github as of August 25, 2011. The issues address some of the current differences between gte and Gambit. These differences may cause loss of Gambit data when an efg or nfg file is opened in the gte tool. Some text within the issues has been updated in this document to provide more context.

Issue 1	**Node name/label not available in gte XML** Node name/label not available in gte XML or gte UI. Notes: can add to XML first, then add to UI at a later time. Moved from kbletzer/standalone/gte/issues. This was formerly issue two in the kbletzer/gte issues list.
Issue 2	**gte does not handle/display iset label** iset label not available in gte - shows Player identification as label for iset. Note: can add iset label to xml, and add UI handling at a later time.
Issue 3	**gte expects a completely unique iset number across all players** gte expects a completely unique iset number per iset regardless of player, whereas Gambit iset numbers are unique per player, but not across players. Currently this is handled in the conversion code by creating a unique iset number for gte. Noting the gap in case it should be handled a different way.
Issue 4	**gte does not handle player payoffs at internal node** gte does not handle player payoffs at internal node. Note: need to update allowed xml structure to handle this, then can add UI handling to gte later.
Issue 5	**gte does not handle more than 2 players in a seamless way** gte does not handle more than 2 players in a seamless way (this was noted by megesdal in to-do list). The behavior is as follows: the file will open, and game tree structure is correct, but payoffs are not displayed for third player, identification of player by unique color doesn't work, although the player is labeled correctly. Gambit currently handles more than 2 players so need to determine how to handle the conversion of a game with more than two players. Note: can fix in XML first, then fix handling for gte UI.
Issue 6	**Sequence Table (right hand side of UI) retains values from previous files in infrequent cases** Sequence Table (right hand side of UI) retains values from previous files in certain (infrequent) situations. For example, load ttt.xml file, then nim.xml file. The ttt sequences are still in the sequence table below the nim sequences. Rechecked issue week of August 22 and it still occurs even in more recent gte version.
Issue 7	**gte does not handle outcome name or outcome label** gte currently does not handle the fields of outcome name or outcome label, which are supported by Gambit. Note: can change first in xml, then in gte UI if necessary.
Issue 8	**gte does not store game description** In the Gambit efg and nfg files there is a game description in the header line of each file. Currently gte does not support the concept of a game "description."

Appendix E: Element/Tag Definition

Element	Children	Parent	Data	Attributes	Notes
gte	(gameName, players, display, (strategicForm | extensiveForm))	N/A		version	root element; takes optional elements of gameName and display parameters, plus required players element. Must have one (only) of either strategicForm or extensiveForm information
gameDescription	no children	gte			name of game - optional note: this does not refer to the file name but could be something like "chain store paradox"; question is where to display it in the UI
players	(player)+	gte			Lists players in order to eliminate any ambiguity regarding player mapping.
player	No children	Players		playerId	Represents an individual player the associated ordering.
display		gte			Elements to be defined at a future time
extensiveForm	node	gte			Information for the extensive game. Information represents the extensive game tree. Has one child element of node, or outcome.

Element	Children	Parent	Data	Attributes	Notes
node	(node|outcome|payoff)*	node, extensiveForm		prob, move, player, iset, isetName, nodeName, outcomeId, outcomeName	Used to build the extensive game tree; can have a parent of child of node, can have a child or outcome.
outcome	(payoff*)	node, extensiveForm		move, prob, nodeName, outcomeId, outcomeName	Outcome - represents a terminal node with or without payoff
payoff	no children	outcome, node	the payoff value	player	Represents the payoff; no children. Has a required attribute of player to indicate who the payoff corresponds to. Can take a single payoff or a payoff matrix.
strategicForm	(strategy*, payoff+)	gte		size	Information for the strategic form game; can have children of payoff (at least one required) or strategy (optional). Has a required attribute of size.
strategy	no children	strategicForm	the strategy labels	player	Valid for strategic games only; used to provide information about the strategy labels. Has a required attribute of player

Element	Children	Parent	Data	Attributes	Notes
payoffs	no children	strategicForm	the payoff matrix	player	Represents the payoff; no children. Has a required attribute of player to indicate who the payoff corresponds to. Can take a single payoff or a payoff matrix.