The xgames package[*]

Benjamin Bernard

August 16, 2023

Abstract

This is a game theory package that places emphasis on user-friendly input and coloring each player's name, actions, and payoffs in a dedicated color for ease of distinction.

1 Introduction

There are a few options out there to create LaTeX renditions of game trees or normal-form games:

- Packages `sgamex` and `egameps` by Martin J. Obsorne.
- Package `istgame` by In-Sung Cho.
- The `trees` or `positioning` libraries of Ti*k*Z.

What sets this package apart from the above is that it provides an integrated package for writing full examples, lecture slides, or assignments questions, in which each player's name, actions, and payoffs are typeset in a dedicated color for ease of distinction. All commands are overlay aware so that game trees, normal-form games, belief spaces, and automata can be revealed gradually in class. Moreover, I placed great emphasis on making the input user-friendly. Finally, there are some gimmicky options like drawing the players' payoffs after a payoff matrix has been parsed or displaying the distribution that a strategy profile induces on a game tree. For a comparison of the different packages' ease of use, see Section 5.

2 Instructions for the Impatient

2.1 Installation

While you could just copy and paste the style files `xgames.sty` and `fikz.sty` into your current working directory, it is neater to install those style files in a local TeXMF root directory so that they are accessible in any working directory. To set up a some folder `mytexmf` as a root directory in MiKTeX, add it to the list of directories in the settings of the MiKTeX console and copy each `package.sty` into the subfolder `mytexmf/tex/latex/package/`. Specifically, for this package place `xgames.sty` and `fikz.sty` into subfolders `mytexmf/tex/latex/xgames/` and `mytexmf/tex/latex/fikz/`, respectively.

2.2 Reading this Documentation Quickly

This package is built on Ti*k*Z and its macros work in any regular `tikzpicture` environment. For best use, the package provides dedicated environments `matrixgame`, `beliefspace`, `gametree`, and

[*]This file has version number v0.92, last revised on August 16, 2023.

automaton that locally redefine some styles and make it easier to customize figures by specifying key-value pairs for the entire figure. All macros form this package and from Ti*k*Z work in any of the dedicated environments, thereby allowing one to create hybrid figures easily.

For a brief introduction on how to implement normal-form games, belief spaces, extensive-form games, and automata, have a look at the examples in Section 3. The examples explain a majority of the package's functionality at a quick glance. The commands and keys are explained following the example in which they appear the first time. In order to tweak those examples to your liking, you can check the full discussion of the commands and their options in Section 4. There is quite a bit of repetition between Sections 3 and 4. The main goal was to reduce search time rather than to produce a short document.

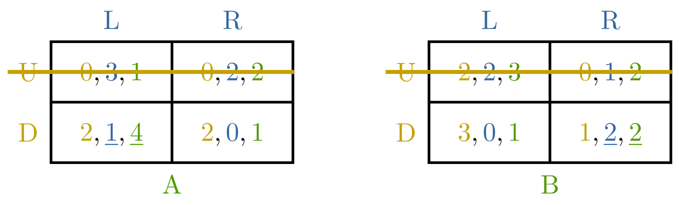
To get started as quickly as possible, load the package with the `nonode` option to improve the compilation speed of game trees by about 10%. With this option, nodes of a game tree are implemented as a circle-shaped path instead of a full-fledged Ti*k*Z node. As a consequence, Ti*k*Z paths ending in such a node will not be shortened, but rather end behind the node's center. The output of all commands from this package will look identical with or without the `nonode` option, but if you intend to mark up the trees with arrows, for example, you may wish to load the package without this option. This documentation, in fact, has been typeset with the `nonode` option.

Error handling is still utterly absent from this package. I apologize for any frustration caused.

3 Examples

3.1 Normal-Form Games and Text Markup

Fig. 1



For Player 1, strategy U is strictly dominated by strategy D. No other pure strategies are iteratively dominated. The pure strategies that survive IESDS are $\{D\}$ for Player 1, $\{L, R\}$ for Player 2, and $\{A, D\}$ for Player 3. There are two pure-strategy Nash equilibria (D, L, A) and (D, R, B).

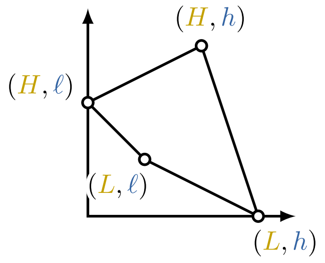
```
\begin{center}
\begin{matrixgame}[top={L, R}, left={U, D}, bottom={A, B}]
  \payoffmatrix{0, 3, 1 & 0, 2, 2\\ 2, \br<3->{1}, \br<4->{4} & 2, 0, 1}
  \payoffmatrix[pos={5,0}]{2, 2, 3 & 0, 1, 2\\ 3, 0, 1 & 1, \br<3->{2}, \br<4->{2}}
  \strike<2->{h}{1}
\end{matrixgame}
\end{center}

For \pl1, strategy $\pl1{U}$ is strictly dominated by strategy $\pl1{D}$. No other pure
↪  strategies are iteratively dominated. The pure strategies that survive IESDS are
↪  $\plset1{D}$ for \pl1, $\plset2{L, R}$ for \pl2, and $\plset3{A, D}$ for \pl3. There
↪  are two pure-strategy Nash equilibria $\ap{D, L, A}$ and $\ap{D, R, B}$.
```

The main macro to typeset normal-form games is `\payoffmatrix`, to which the input is provided as in a regular `tabular` environment. The key `pos={5,0}` offsets the center of the second payoff matrix by $(5, 0)$ in the Ti*k*Z coordinate system that underlies the `matrixgame` environment. Labels provided as a comma-separated list to the keys `top`, `left`, and `bottom` are added to all payoff matrices of this `matrixgame`. The `\strike` command strikes out a player's strategy in all payoff matrices. Here, `h` in the first argument stands for horizontal. With arguments `v` and `x`, strategies of players 2 and 3 can be struck through. Best responses are highlighted using `\br`. Both `\strike` and `\br` are overlay aware.

The commands `\pli` for $i = 1, \ldots, 9$ color any text in player i's color.[1] The color palette can be changed by `\setplayercolors` or `\letplayercolors`. For example, the preamble of this file contains the line `\setplayercolors{HTML}{c4a100; 3465a4; 4f9905; cf5c00}` to match the "tango" color scheme of the `minted` package. Without argument, the command `\pli` prints player i's name, which is "Player i" by default. The commands `\plseti` for $i = 1, \ldots, 9$ color the elements of a comma-separated list in player i's color. Action profiles or payoff vectors can be colored by passing the player's actions or payoffs, respectively, as a comma-separated list to `\ap` (short for action profile).

Fig. 2

$$\begin{array}{cc|c|c|}
 & & \multicolumn{2}{c}{\text{Customer}} \\
 & & h & \ell \\
\hline
\text{Firm} & H & 2, \underline{3} & 0, 2 \\
\hline
 & L & \underline{3}, 0 & \underline{1}, \underline{1} \\
\hline
\end{array}$$



A long-lived Firm chooses to exert a (H)igh or a (L)ow amount of effort and the short-lived Customer chooses to buy a (h)igh-priced or a (ℓ)ow-priced product.

```latex
\begin{center}
\begin{matrixgame}[force math, top={h,\ell}, left={H,L}, player names={Firm,Customer}, br]
  \payoffmatrix[w=1.3, h=0.75]{2, 3 & 0, 2\\ 3, 0 & 1, 1}
  \drawpayoffs<2->[scale=0.75, shift={5,-0.8}]
\end{matrixgame}
\end{center}

\definevar{H, L; h, \ell}
A long-lived \pl1 chooses to exert a ($\H$)igh or a ($\L$)ow amount of effort and the short-lived \pl2 chooses to buy a ($\h$)igh-priced or a ($\ell$)ow-priced product.
\releasevar
```

The key `force math` places action labels in all payoff matrices and drawn payoffs in math mode. The key `player names={Firm, Customer}` changes the players' names globally and adds the players' names to the payoff matrix. When the key `br` is passed to `\payoffmatrix`, all best responses are highlighted automatically with `\br`, given that payoffs are explicit numbers and there are three or fewer players. The keys `w` an `h` govern the width and height of the cells, respectively. The command `\drawpayoffs` draws the pure-action stage game payoffs of a two-player game parsed with `\payoffmatrix` as well as their convex hull. The key `scale=<factor>` scales the drawn image by `<factor>` and the key `shift={x,y}` shifts the origin by (x, y) from the center of the payoff matrix.

Instead of writing `\pli{<text>}` to refer to player i's action `<text>`, one can also define commands `\<text>` using `\definevar`. Players are separated by semicolons and each player's actions by commas.

[1]Formally, the macro is a single-argument macro `\pl{<num>}` that scans for a second argument with a `\futurelet` command. We can omit the braces around `<num>` because it is a single token.

If any existing macros are overridden by `\definevar`, they are restored by `\releasevar`. Below four examples show four more ways, with which one can quickly colorize text by players.

If the Agent's action $\alpha \in \mathcal{A}$ is observed by the Principal, the Principal can incentivize the Agent to choose his/her preferred action α_{*} by making the payment conditional on the observed action.

```
\setplayernames{Principal, Agent}
\definevar[player=2]{\alpha, \A->\mathcal{A}}
If the \pl2's action $\alpha \in \A$ is observed by the \pl1, the \pl1 can incentivize the
↪  \pl2 to choose his/her preferred action $\alpha_{\ast}$ by making the payment
↪  conditional on the observed action.
\releasevar{\alpha, \A}
```

If the variable to be defined is a command sequence, the content of that command sequence is typeset in the player's color. One can set the variable to any arbitrary content with the notation `\variable-><content>`. Note that subscripts are absorbed in the color of the variable as it is interpreted as different instance of the same variable. By setting the key `player=<num>`, the first variable is parsed as player `<num>`'s variable and the player counter increases with each semicolon. In this instance, the same effect could be achieved by `\definevar{; \alpha, \A->\mathcal{A}}`. If one wishes to release only some, but not all variables defined by `\definevar`, one can pass a comma-separated list to `\releasevar` of variables to be released. Doing so is marginally slower than releasing all variables at once and certain auxiliary macros will not be released (such as `\qhat` in below example). It is thus recommended to simply release all variables at once at the end of an example.

Since $u_1'(\hat{q}_1, \hat{q}_2) > 0$, Firm 1 has an incentive to deviate and produce a quantity $q_1' > \hat{q}_1$.

```
\setplayernames{Firm~1, Firm~2}
\multivar{u}
\multivar[prefix=hat, postfix=']{q}
Since $\u1'(\qhat1, \qhat2) > 0$, \pl1 has an incentive to deviate and produce a quantity
↪  $\q1' > \qhat1$.
\releasevar
```

The command `\multivar` defines macros `\<text>i` for $i = 1, \ldots, 9$ which typeset `<text>_i` in player i's color. It is possible to refer to different instances of the same variable by setting the `prefix` and the `postfix` keys. The key `prefix=<prefix>` defines macros `\<text><prefix>i`, which are typeset as `\<prefix>{<text>}_i`. Instead of `hat`, one could use it to typeset `tilde`, `bar`, `underline`, `mathcal`, etc. To denote different instances by superscripts or primes, set the `postfix` key to `^`, `'`, or `{^,'}`, respectively. Contrary to subscripts, superscripts or primes are not absorbed in the player's color by default because they could signify powers or derivatives. Again, `\releasevar` restores any macros that have been overwritten, including hats, bars, tildes, etc.

The Incumbent and the Challenger choose a campaign budget β_I and β_C, respectively. The probability of winning depends on the entire profile $\beta = (\beta_I, \beta_C)$.

```
\setplayernames{Incumbent, Challenger}
\multivar[index=initial]{\beta}
The \pl1 and the \pl2 choose a campaign budget $\beta1$ and $\beta2$, respectively. The
↪  probability of winning depends on the entire profile $\beta0 = (\beta1, \beta2)$.
\releasevar
```

By passing the key-value pair `index=initial` to `\mutlivar`, the player's actions are indexed by the first letter of their names instead. By passing player number 0, coloring and subscripts are suppressed. Alternatively, the same effect could be achieved by declaring `\multivar[index=initial]{\b->\beta}` before the example and by writing `\beta = (\b1, \b2)` in the example.

Players $i = 1, 2$ choose $a_i \in$ {Rock, Paper, Scissors}. Player 1's best response is $\mathcal{B}_1$(Rock) = {Paper}.

```
\resetplayernames
Both players~$i = \apm{1, 2}$ choose $a_i \in \textset{Rock, Paper, Scissors}$. \pl1's best response is $\BR[1](\pltext2{Rock}) = \textset1{Paper}$.
\releasevar
```

The players' names can be reset to default with `\resetplayernames`. The command `\apm` colors a comma-separated list by player colors as displayed in a `\payoffmatrix` (hence the `m` in `\apm`). Actions and sets of actions that are spelled out, rather than abbreviated by variables, can be typeset with the commands `\pltexti`, `\textset`, and `\textseti` for $i = 1, \ldots, 9$. The command `\BR[i]` is intended for typesetting player i's best response correspondence.

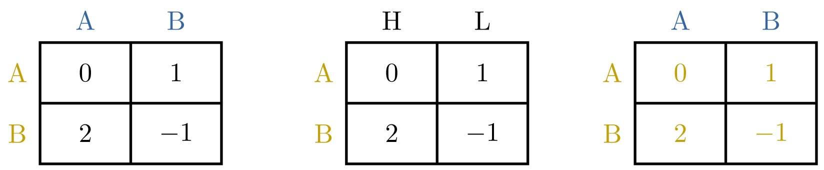


Fig. 3

```
\matrixsetoptions{w=1.2}
\begin{matrixgame}[top left={A, B}]
  \payoffmatrix{0 & 1 \\ 2 & -1}
\end{matrixgame}
\matrixsetoptions{single player=nature}
\begin{matrixgame}[top={H, L}, left={A, B}]
  \payoffmatrix{0 & 1 \\ 2 & -1}
\end{matrixgame}
\begin{matrixgame}[top left={A, B}, zero-sum]
  \payoffmatrix{0 & 1 \\ 2 & -1}
\end{matrixgame}
```

The key `single player` determines how the game is colored when payoffs of only a single player are displayed. Shown above are three matrices for the values `default`, `nature`, and `zero-sum`, respectively. Those values can be set locally, for a single `matrixgame` environment or globally through the command `\matrixsetoptions`. Alternatively, the colors of the cell entries and labels can be customized with the `color` or `player` keys; see Section 4.5 for further examples.

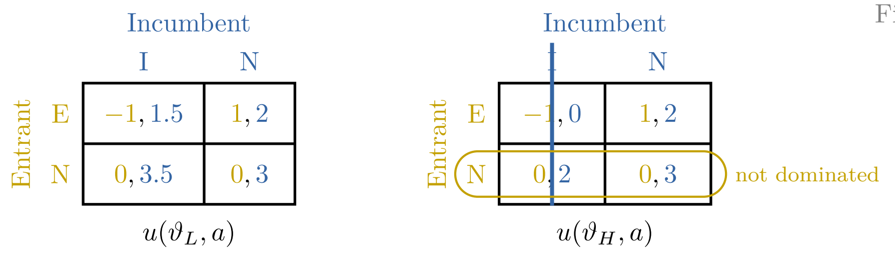


Fig. 4

```
 1 \setplayernames{Entrant, Incumbent}
 2 \begin{matrixgame}[name=startup, players, w=1.4, strike width=1.5pt, immediate]
 3   \payoffmatrix[w={1.6, 1.2}]{-1, 1.5 & 1, 2 \\ 0, 3.5 & 0, 3}
 4   \payoffmatrix[pos={5.5, 0}]{-1, 0 & 1, 2 \\ 0, 2 & 0, 3}
 5   \toplabel{I, N}
 6   \leftlabel{E, N}
 7   \bottomlabel[list sep=;]{$u(\vartheta_L, a)$; $u(\vartheta_H, a)$}
 8   \strike<2->[2]{v}{1}
 9   \draw<3->[player1, thick, rounded corners=8.5] (3.52, -0.7) rectangle (7.1, -0.1);
10   \node<3->[player1, right, font=\footnotesize] at (7.1, -0.4) {not dominated};
11 \end{matrixgame}
```

If the labels are passed to the `matrixgame` as key-value pairs, they are typeset at the end of the environment to ensure that all payoff matrices have been parsed. Since the `\strike` and the `\drawpayoffs` commands require access to those labels, their output is drawn only at the end of the environment by default, regardless of when their input is parsed. In order to mark up the image above any `\strike`s, we use the `immediate` key. This key causes the commands `\strike` and `\drawpayoffs` to be typeset as they are parsed. Passing this key necessitates that:

- Top, left, and bottom labels are issued with their separate commands *after* all payoff matrices have been parsed and *before* any `\strike` or `\drawpayoffs` commands are issued.
- If the key `br` is set, it is set for an individual `\payoffmatrix` of at most two players (because best responses by player 3 cannot be determined matrix by matrix).

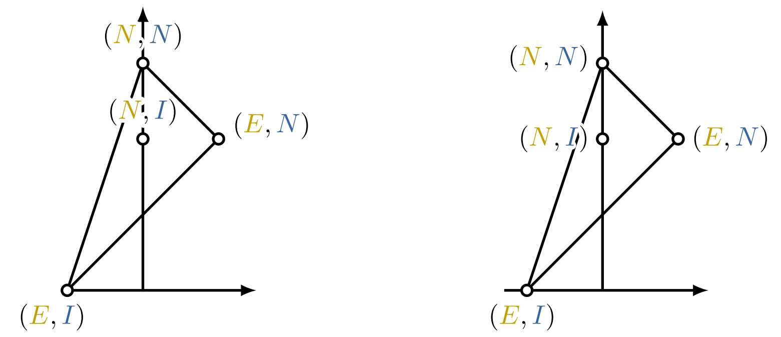
The key `immediate` is always active in regular `tikzpicture` environments. Any environments of this package can be marked up by any Ti*k*Z commands such as the `\draw` and `\node` commands on lines 9–10. In such commands, player i's color can be accessed by the color `playeri`.

The `\strike` width and many other options can be customized by passing them as environment options. See Section 4 for a comprehensive list. `\strike` takes an optional argument `<number>` if a strike is to be drawn in only one matrix. If the players' names have been defined before the `matrixgame` environment, such as through `\setplayernames{<list of names>}` before the `center` environment here, you can display those names with the `players` key instead of setting them again with the `player names` key as in Figure 2. Column widths can be adjusted individually with the key `w={<list of widths>}`. The key `h={<list of heights>}` individually adjusts the height of rows.

The command `\bottomlabel` behaves differently for 2-player and 3-player games. In 2-player games, `\bottomlabel` is interpreted as a description of states rather than the actions of a third player. It is thus unaffected by the `force math` key, it is typeset in the package foreground color `xg-fg` by default, and it is separated slightly further from the `\payoffmatrix` than player 3's action labels would be. In order to attain the same label distance as from payoff matrices in 3-player games, pass the key `tight`. The list separator of any label command can be changed with the `list sep` key. This is convenient if any of the labels contain a comma, the default separator of lists. If the `list sep` key is passed at the environment level, it is changed for all labels, including labels set through key-value pairs.

3.2 Payoff Sets

Fig. 5

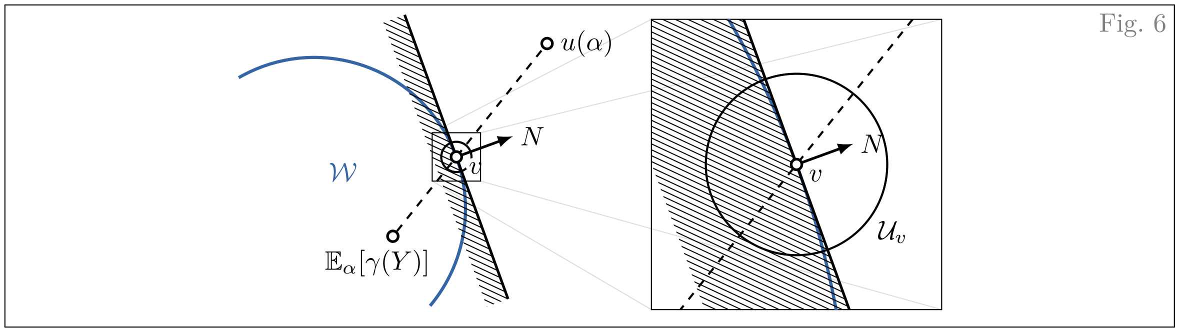


```
\begin{tikzpicture}
  \drawpayoffs[name=startup, matrix=2, force math, axis=1.25]
\end{tikzpicture}
\begin{tikzpicture}
  \draw[edge, -latex] (-1.3, 0) -- (1.4, 0);
  \draw[edge, -latex] (0, 0) -- (0, 3.7);
  \state (EN) at (1, 2) <2->{$\ap{E, N}$};
  \state[label=260] (EI) at (-1, 0) {$\ap{E, I}$};
  \state[label=175] (NN) at (0, 3) {$\ap{N, N}$};
  \state[left, contour=delay] (NI) at (0, 2) {$\ap{N, I}$};
  \draw[edge] (EN) -- (NN) -- (EI) -- (EN);
\end{tikzpicture}
```

To align payoff sets vertically centered with the matrixgame that defines the payoffs, it is easiest to draw payoffs in a separate `tikzpicture` environment rather than in the same `matrixgame`. You can store any parsed `matrixgame` with the option `name=<text>` and recall it with `name=<text>` in `\drawpayoffs`. The key `matrix=<number>` determines which `\payoffmatrix` of the environment is drawn. Without the `name` key, the most recent unnamed `matrixgame` is drawn.[2] If payoffs are drawn in a separate environment, the key `force math` needs to be re-issued. The key `axis=<factor>` determines how far beyond the maximum payoffs the axes extend. Labels of each payoff pair are drawn on the opposite side of the center of mass among payoff pairs. This will look good in many repeated games we teach, but in some games (as above) it may not. Labels can be turned off with the key `labels=off`.

It is not difficult (and instructive) to draw this payoff set manually. The syntax of the `\state` command is identical to a Ti*k*Z `\node` with two exceptions: braces are optional if no label is to be set and a separate overlay instruction `<overlay>` can be given to the label. The `label` key determines the angle in degrees, at which the label is set from the node. The eight standard directions `left`, `above left`, `above`, etc. are also supported—either as value to the `label` key or as its own key. The `contour` key accentuates the label from the background using the *contour* package. The contour color is the package background color `xg-bg`. You can change the color with any means provided by the *xcolor* package. Typically, any `\draw` command issued after a `\state` command is drawn on top of the state and its label. By passing the value `delay` to the `contour` key, the label is typeset at the end of the `tikzpicture`. This may be convenient because the coordinate can be used already. Lines with the same `line width` as the `\state`s can be drawn with the `edge` Ti*k*Z style.

[2]In this instance, the payoff pairs of the first two players in the second matrix parsed in Figure 1 would be drawn. Labels, however, would be taken from the product-choice game because they are overwritten.



```
\halfspacesetoptions{depth=0.3, hatch=0.1}
\begin{tikzpicture}
  \zoom<3->[l={-0.32, -0.32}, u={0.32, 0.32}, scale=6, shift={4.5, -0.1}]{
    \coordinate (v) at (0, 0);
    \draw[very thick, player2] ($(v) + (20:-2) + (120:2)$) arc(120:-40:2);
    \node[player2] at (-1.5, -0.2) {$\mathcal{W}$};
    \halfspace<2->[w=4, label=0] at (v) dir (20:0.8) {$N$};
    \state (u) at (1.2, 1.5) {$u(\alpha)$};
    \state[label=-120] (gamma) at ($1.7*(v) - 0.7*(u)$) {$\mathbb{E}_{\alpha}[\gamma(Y)]$};
    \draw[thick, dashed] (gamma) -- (u);
    \draw<4->[thick] (v) circle (2mm);
    \state[contour, label=-30] at (v) {$v$};
  }{
    \node at ($(v) + (-35:2.6mm)$) {$\mathcal{U}_v$};
  }
\end{tikzpicture}
```

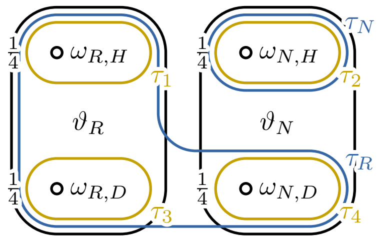
The lower half-space through `<coord>` with normal vector `<normal>` is visualized with the command `\halfspace«overlay»[<options>] at (<coord>) dir (<normal>) {<label text>};`, where both `<coord>` and `<normal>` can be any Ti*k*Z coordinates. The normal vector is labeled by `<label text>` in the direction `<dir>` specified by the key-value pair `label=<dir>`. If `<normal>` consists only of an angle, rather than a polar coordinate, the normal vector is suppressed. The keys `w` and `depth` specify the width and the depth, respectively, of the half-space and the key `hatch` specifies the distance between two lines of the hatching pattern. The keys can be set globally with the `\halfspacesetoptions` command.

The command `\zoom«overlay»[<options>]{<figure>}{<markup>}` displays both the `<figure>` and a zoom-in of it, in which the zoomed-in part is marked up by `<markup>`. The zoom-in is shown only on the slides specified by `<overlay>`. The keys `l={<x>,<y>}` and `u={<x>,<y>}` specify the lower left and the upper right corner of the rectangular area that is zoomed in by a factor of `scale=<factor>`. Finally, they key `shift={<x>,<y>}` shifts the center of the zoomed-in part by `(<x>,<y>)` in the non-zoomed-in coordinate system. Note that the size of nodes, the density of the hatching pattern, or the length of the visual representation of the normal vector are unaffected by the zoom.

The `\zoom` command is incompatible with any `delay` options. Because coordinates are defined both in the entire figure and the zoom-in, any delayed commands are executed only in one of the coordinate systems. The original coordinates are restored after the `\zoom` command (and hence any delayed features will be missing from the zoom-in). Because delayed features are not supported, the `nonode` package option is locally disabled within a `\zoom` command.

3.3 Belief Spaces

Fig. 7



```
1  \begin{beliefspace}[radius={3.2mm, 3.9mm}, event=4.6mm, right=-1pt]
2    \foreach[count=\i] \x in {R, N} {\foreach[count=\j] \y in {H, D} {
3      \pgfmathtruncatemacro{\k}{(\j-1)*2+\i}
4      \state (w\k) at (2.2*\i, -1.4*\j) {$\omega_{\x, \y}$};
5    }}
6    \event (w1) -- (w3) {$\vartheta_R$};
7    \event (w2) -- (w4) {$\vartheta_N$};
8    \informset<3->[player=2, label=30, contour] (w4) -- (w3) -- (w1) -- (w3) {$\tau_R$};
9    \informset<3->[label=30, contour] (w2) {$\tau_N$};
10   \xforeach \i in {1, ..., 4} {
11     \informset<2->[player=1, label=-30, contour=delay] (w\i) {$\tau_{\i}$};
12     \informlabel<4->[contour] (180:1) at (w\i) {$\frac{1}{4}$};
13   }
14 \end{beliefspace}
```

The two main commands in a `beliefspace` environment are `\state`, to denote states of the world, and `\informset`, to indicate information sets or events. The command `\event` is just a shorthand for `\informset[player=0]`. All three commands are overlay aware. The `\state` command is as described before, except that the label is always set to the right within a `beliefspace` environment. This ensures that information sets can be drawn nicely around the state and its label. The necessary width of information sets is measured automatically. However, because bounding boxes are rectangular but letters are not, some manual adjustment may be necessary. The key-value pairs `left=<dimen>` and `right=<dimen>`, respectively, add `<dimen>` to the left and right of the measured width.

The `\informset` command takes a list of states, nodes, or coordinates in counterclockwise order, separated by `--`, and draws an information set around those states with radius specified by the `radius` key of either the `beliefspace` environment or the `\informset` command. It is possible to pass by the same state multiple times as on line 8 to display non-convex information sets. The first state, however, has to be a state on the boundary of the convex hull. If the `radius` key is passed to the `beliefspace` environment, the `radius` key takes a comma-separated list of radii for the different players. The `event` key sets the radius of player 0 (nature). The key `player=i` of the `\informset` commands sets the active player to player i, which sets the default radius and color of the current and future information sets those of player i. A label can be provided to information sets within braces after the last coordinate, and it can be placed by the `label` key. For non-singleton information sets, the default position is in the center of the information set. For singleton information sets, the default position is to the right of the information set. If a label direction is provided, it is offset at that angle from the first state appearing within the `\informset` command.

Additional labels can be added to states with `\informlabel`. The offset `180:1` causes the label to be placed to the left of `(w\i)` at the radius of player 1. Alternatively, the offset can be specified as a regular polar coordinate `<direction>:<dimen>` or as the triple `<direction>:<player>:<dimen>`. In the latter case, the label's anchor is placed at the radius of player `<player>` plus `<dimen>`.

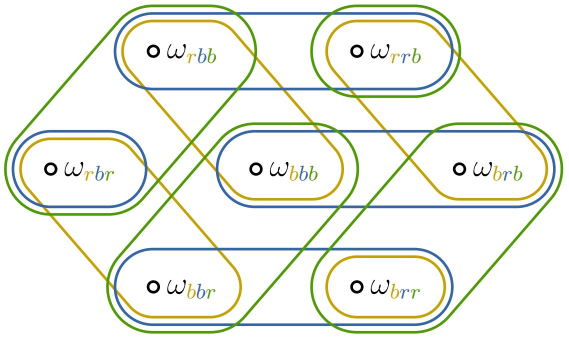


Fig. 8

```
\informsetradius{4mm, 5mm, 6mm}
\begin{beliefspace}[center, font=\large]
  \state (w0) at (0, 0) {$\omega_{\apm[]{b,b,b}}$};
  \xforeach[count=\i] \x in {{r,b,b},{r,r,b},{b,r,b},{b,r,r},{b,b,r},{r,b,r}} {
    \state (w\i) at (180-60*\i:2.7 and 1.8) {$\omega_{\noexpand\apm[]{\x}}$};
  }
  \foreach \i in {1, 2, 3} {
    \pgfmathtruncatemacro{\idx}{2*\i-1}
    \informset[player=\i] (w\idx) -- (w0);
    \pgfmathtruncatemacro{\j}{Mod(2*\i+1,6)+1}
    \informset (w\j);
    \foreach \s in {+, -} {
      \pgfmathtruncatemacro{\j}{Mod(\idx\s1-1,6)+1}
      \pgfmathtruncatemacro{\k}{Mod(\idx\s2-1,6)+1}
      \informset (w\j) -- (w\k);
  }}
\end{beliefspace}
```

Instead of specifying the radii of the players' information sets in each `beliefspace` environment, one can set them globally with the `\informsetradius` command. If labels are unequally long, they can be centered horizontally within the information sets with the `center` key. Doing so requires measuring the widths of all labels, hence typesetting the labels is postponed until the end of the environment. To ensure that values of the running variables in the for loops are retained, the package provides a version `\xforeach` of pgffor's `\foreach`, within which each label is fully expanded. Without the expansion, all labels would be set to ω_{rbr} since `\x` is equal to `{r,b,r}` at the end. Within an `\xforeach` loop, each non-expandable macro that appears in a label (such as `\apm` here) needs to be protected by immediately preceding it with a `\noexpand`.[3] Alternatively, one can directly pass the length of the widest label with the key-value pair `center=<dimen>` and use a `\foreach` loop. Finally, the optional argument `<sep>` of `\apm` indicates with which separator `<sep>` the attributes of the different players are typeset.

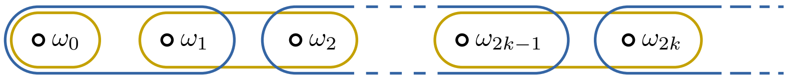


Fig. 9

[3]Roughly speaking, any macro that directly produces a legible character is expandable, such as Greek letters, `\int`, `\oplus`, etc. Non-expandable macros are procedural commands that change instructions on how something is typeset, such as `\frac`, `\mathcal`, `\color`, or `\apm` in this example.

```
\begin{beliefspace}[identical width=false]
  \foreach \i in {0, 1, 2} {\state (w\i) at (1.7*\i, 0) {$\omega_{\i}$};}
  \foreach \i/\x in {3/2k-1, 4/2k} {\state (w\i) at (2.2*\i-1, 0) {$\omega_{\x}$};}
  \coord (w5) at (9.5, 0);
  \informset[player=1] (w0);
  \informset (w1) -- (w2);
  \informset (w3) -- (w4);
  \informset[player=2] (w0) -- (w1);
  \informset (w2) .. (w3);
  \informset (w4) -. (w5);
\end{beliefspace}
```

A countably infinite belief space can be drawn by separating coordinates with `..`, `-.`, or `.-` instead of `--`. In the first case, the center portion of the information set is dashed. In the latter two cases, a "trailing off" pattern is drawn towards the side with the dot. One of the bounding semi-circles is not drawn, but the information sets are otherwise of the same size. Here, the half-open information set is drawn around coordinate `(w5)`, which is defined using the command `\coord`. It is similar to the Ti*k*Z-native `\coordinate`, but additionally takes care of background behavior needed in belief spaces.

By default, information sets are drawn as if the labels of all states had the same width. This creates a uniform look if the states appear in a grid, but it may look odd if the labels of some states are much longer than those of other states. This behavior can be turned off with the key `identical width=false`. This option is currently only supported if all states are in a horizontal line.

```
\begin{beliefspace}[state=false]
\stategrid[enum=grid]{1 & 3 & 2\\ 3 & 1 & 2\\ 2 & 2 & 1}
\foreach \i in {1, 2, 3} {
  \informset[player=1] (w\i1) -- (w\i3);
  \informset[player=2] (w1\i) -- (w3\i);
}
\end{beliefspace}
```

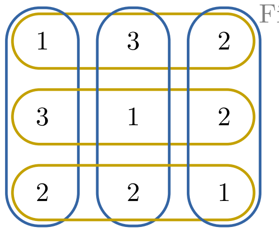


Fig. 10

A rectangular grid of states can be initialized with the `\stategrid` command, whose argument is formatted as in a `tabular` environment. The key `enum=grid` assigns coordinate `(w<row><col>)` to the state in row `<row>` and column `<col>`. Without this key-value pair, the coordinates are labeled consecutively from `(w1)` to `(w9)` instead. If the `state` key at the environment level is set to `false`, the state is represented directly with its label rather than through a circular node.

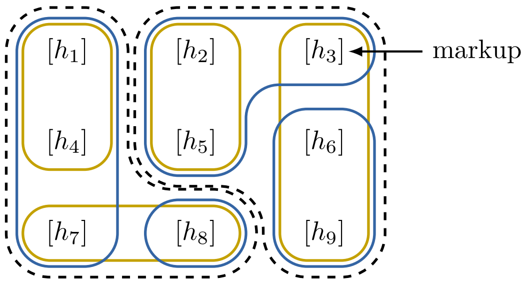
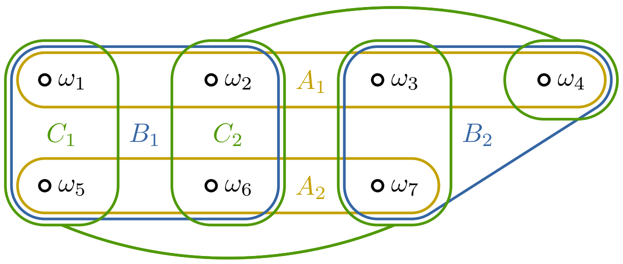


Fig. 11

```
\begin{beliefspace}[event=5.8mm, state=false]
  \stategrid[name=h, h=1.7, v=1.2, autolabel=$\lbrack h_{\enum}\rbrack$]{3\\ 3\\ 3}
  \foreach \i/\j in {1/4, 2/5, 3/9, 7/8} {\informset[player=1] (h\i) -- (h\j);}
  \informset[player=2] (h1) -- (h7);
  \informset (h2) -- (h5) -- (h2) -- (h3);
  \informset (h6) -- (h9);
  \informset (h8);
  \event[line=dashed, sticky] (h1) -- (h7) -- (h8) -- (h7);
  \event (h2) -- (h5) -- (h6) -- (h9) -- (h3);
  \draw[thick, latex-] (h3) -- ++(1.3, 0) node[right] {markup};
\end{beliefspace}
```

The key `name=h` assigns coordinates `(h1)` through `(h9)` to the states in the grid. The keys `h` and `v` determine the horizontal and vertical spacing, respectively. The states are labeled automatically through the `autolabel` key. The macro `\enum` is replaced with the index of the coordinate (which is `<row>, <col>` if the `enum` key is set to `grid` instead).[4] When the `autolabel` key is set, the mandatory argument of `\stategrid` specifies the number of states in each row, separated by `\\`.

Fig. 12



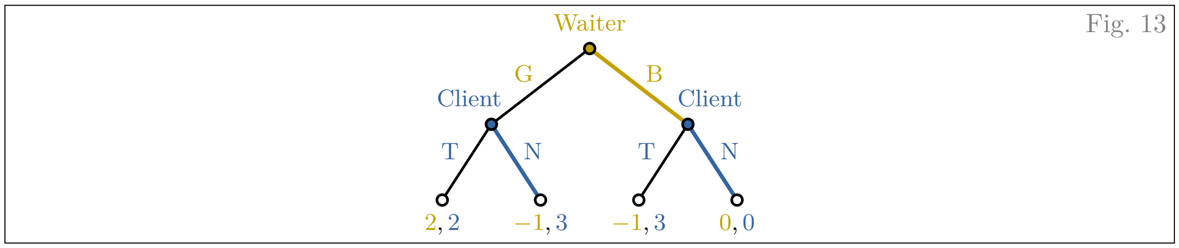
```
\begin{beliefspace}
  \stategrid[h=2.2, v=1.4, autolabel=$\omega_{\enum}$]{4\\ 3}
  \informset[player=1] (w1) -- (w4) {$A_1$};
  \informset[label=0.8] (w5) -- (w7) {$A_2$};
  \informset[player=2] (w1) -- (w5) -- (w6) -- (w2) {$B_1$};
  \informset[label=square] (w3) -- (w7) -- (w4) {$B_2$};
  \informset[player=3] (w5) -- (w1) -- (w5) ~~ (w7) -- (w3) -- (w7) {$C_1$};
  \informset (w2) -- (w6) -- (w2) ~^~ (w4) {$C_2$};
\end{beliefspace}
```

By default, the `label` of an information set is placed at the barycenter of the coordinates in the information set. If a value `<value>` between 0 and 1 is passed to the `label` key, the label is placed at a fraction `<value>` of the way from the first coordinate to the last coordinate. If the value `square` is passed to the `label` key, the label is placed in the middle between the highest and lowest x- and y-coordinates.

Information sets containing a disjointed set of states can be drawn with the separator `~~`. By default, disjointed sets of states are connected by an arc. See Figure 34 for the same figure with the alternative `exterior=extend` option. Note that the coordinate list will contain duplicate coordinates at points of non-convexity as it is the case in Figure 7. The default label placement for such information sets is `square` in the first connected set of states. With the separator `~~`, the arc is drawn opposite the previous/next coordinate in the list, i.e., opposite of `(w1)` at `(w5)` and opposite of `(w3)` at `(w7)`. This requires that each connected set of states contains at least two states. For singleton states, you must provide the direction of the extension manually with `~^~`, `~v~`, `~<~`, and `~>~` for the arc to be drawn above, below, to the left, and to the right, respectively.

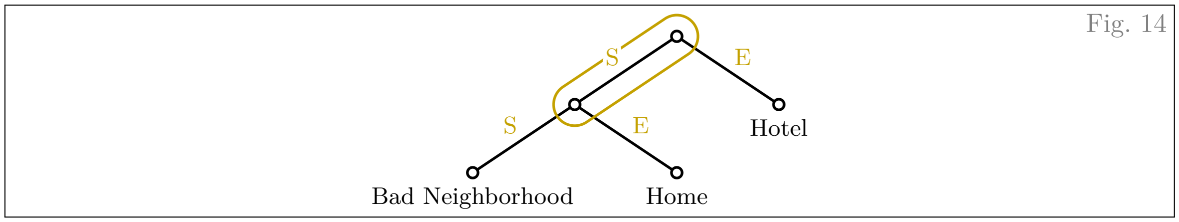
[4] I use `\lbrack` and `\rbrack` instead of `[` and `]` here because nested brackets pose issues both for parsing optional arguments and the `minted` package, which is used to color the code snippets in this documentation.

3.4 Extensive-Form Games



```
\begin{gametree}[player names={Waiter, Client}]
  \branch[player=1, root=w] from (0, 0) to[h=2.6, v=1] {G; B<3->};
  \branch[player=2, label=130] from (w1) to[h=1.3] {T(2,2); N<2->(-1,3)};
  \branch[label=50] from (w2) to {T(3,-1); N<2->(0,0)};
\end{gametree}
```

We can draw game trees in the `gametree` environment by using the `\branch` command with syntax `\branch«overlay»[<opt>] from (<parent>) to[<edge opt>] {<list of children>};` The key `player=i` labels the parent of the branch with player *i*'s name and causes the action labels, the parent's interior, and any highlighted actions to be colored in player *i*'s color. The key `root=<name>` defines a coordinate `(<name>)` at the parent node. The `root` key must be set if `(<parent>)` is not a named coordinate. The key `label=<angle>` places the center of the label at angle `<angle>`.[5] In this tree, the key `v` determines the vertical offset of the children from the parent node and the key `h` determines the horizontal distance between children. Both values are stored until overridden. The list of children is provided as a semi-colon-separated list, where each child node is described with the syntax `<action>«highlight»[<child opt>](<payoffs>)`. Here, `<action>` is the label of the edge in the tree, `<highlight>` is an overlay specification, causing the edge to be highlighted in player *i*'s color on the specified slides, `<child opt>` allows fine-tuning of individual edges and their labels, and `<payoffs>` will generate a terminal node with label `\ap{<payoffs>}` if `<payoffs>` is a comma-separated list and label `<payoffs>` otherwise. By default, a coordinate `(<parent>i)` is defined at the position of child *i* in the list, which can be used as the parent of future `\branch`es.



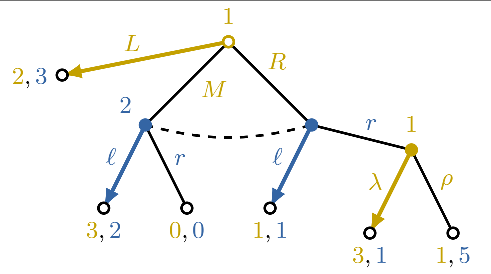
```
\begin{gametree}[index=action, nodes=clear]
  \branch[player=1, root=w] from (0, 0) to[h=3, v=1, contour=delay] {S; E(Hotel)};
  \branch from (wS) to {S(Bad Neighborhood); E(Home)};
  \informset[label=none] (w) -- (wS);
\end{gametree}
```

With the key `index=action`, the coordinate at child *i*'s location is called `(<parent><action i>)` instead of `(<parent>i)`. This works only if `<action>` is plain text, not if it contains any Greek letters, any macros, or any other tokens not allowed in Ti*k*Z coordinate names. The key `nodes=clear` is

[5]This is the default placement algorithm `center`. You can select placement algorithms `simple` and `box` with the key `label placement` at the `\branch` level or with the key `node label placement` at the environment level.

shorthand for `nodes=empty` and `node labels=none`, which suppresses coloring and labeling of any non-terminal nodes in the tree; see below example for a full description of the `nodes` key. The `contour` key draws a contour around the labels of the edges using the `contour` package. With the value `contour=delay`, the edge labels are drawn at the end of the `tikzpicture`. This is convenient in this instance because the `\branch` command defines the coordinates around which `\informset` is placed, hence it has to appear before the `\informset` command.

Fig. 15



```
1 \treesetoptions{labels=short, force math, path=arrow}
2 \begin{gametree}[scale=1.1, nodes=full, root=hollow]
3   \branch[player=1, root=w, payoffs=186] from (0, 0) to[v=-0.4, h=-2] {L<1->(2,3)};
4   \branch from (w) to[v=1, h=2] {M[1,0.4]; R};
5   \branch[player=2, label=none] from (w2) to[h=1] {\ell<1->(3,2); r(0,0)};
6   \branch[offset={0.35,0}] from (w3) to[v={1,0.3}, h=1.7] {\ell<1->(1,1); r};
7   \informset[curved=15] (w2) -- (w3);
8   \branch[player=1] from (w32) to[v=1, h=1] {\lambda<1->(3,1); \rho(1,5)};
9 \end{gametree}
```

The key `labels=short` abbreviates player labels of nodes and information sets to the player number. The key `force math` puts all edge labels into math mode. The key `path=arrow` highlights chosen actions with arrows. Since all three keys are passed to `\treesetoptions`, those will be the default values for all future `gametree` environments. If the keys were passed to the environment instead, their values would be the default only for that `gametree`. The key `scale=1.1` is passed to the underlying Ti*k*Z image and scales the entire environment by a factor of 1.1.

The `nodes=full` key changes the style of non-terminal nodes in the tree so that their outlines are colored with the active player's color as well. Other admissible node styles are `default`, `empty`, and `hollow`. Those styles can be passed to the `node` key of an individual `\branch` to stylize only the parent node of that branch. The root of the tree can be stylized separately by setting the `root` key globally or at the environment level.[6] Note that the `root` key serves a different purpose at the branch and at the environment level. To reset separate styling of the root, use `root=clear` globally or at the environment level. Node styles are overlay aware; see Figure 19 for an example.

By default, children are aligned symmetrically about the parent node. There are two ways to draw asymmetric branches. The first method is to simply use two `\branch` commands as on lines 3–4 of the code. Note that the `root` key need not be repeated. The behavior of keys `v` and `h` is special if a branch contains only a single child. Then the child is simply offset by `(v,h)` from the parent node. The second method is to provide comma-separated lists to the `v` and `h` keys. For such input, the vertical offset of the children from the parent node and the horizontal distance between children, respectively, cycles through these lists. In a vertically oriented tree, the children are horizontally centered about an `offset` from the parent node, which can be set with the key-value pair `offset={<x>,<y>}`.

[6]The main purpose of `root` key is to set the root style globally by passing it to `\treesetoptions`. In this example—for an individual `gametree`—the same effect can be achieved by passing the `node=hollow` key to the root `\branch`.

With the key `payoffs=<angle>`, the payoff labels of terminal payoffs from this branch are set at an angle `<angle>`. The options `[1,0.4]` passed to action label `M` cause the label to be placed 90° counterclockwise offset at 40% of the way from parent to child. `l` here stands for "left in the direction from parent to child" and key `r` would place the label on the "right side in the direction from parent to child". The key `curved=<curv>` of the `\informset` command causes an information set to be drawn as a dashed line with curvature `<curv>`. The default curvature is `20`. A piecewise straight dashed line connecting all nodes in the information set can be obtained with the `dashed` key. The default placement of the label of a dashed information set is `above left` the first node in the information set. To avoid duplicate placement of labels from nodes, we set the key `label=none` on line 5, which remains active until the active player changes by the next invocation of the `player` key.

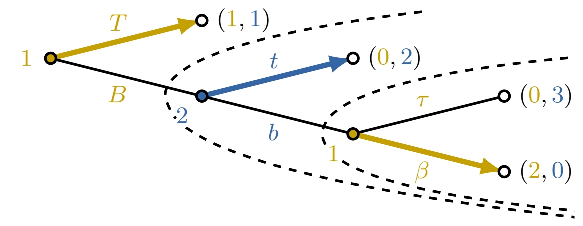


Fig. 16

```
\begin{gametree}[horizontal, terminal=p, subgame={depth=9.3mm}, alert width=2pt]
  \branch[player=1, root=w] from (0,0) to[v=1, h=2] {T<4->(1,1); B};
  \branch[player=2, label=225, contour=delay, very sticky] from (w2) to {t<3->(0,2); b};
  \branch[player=1] from (w22) to {\tau(0,3); \beta<2->(2,0)};
  \subgame<2->[inner sep=2.7ex, w=0.62] at (w22);
  \subgame<3->[inner sep=3.2ex] at (w2);
\end{gametree}
```

The key `horizontal` orients the tree horizontally and switches the behavior of `h` and `v`. The key `terminal=p` adds parentheses around the terminal payoffs. The key `alert width` governs the width of the highlighted edges and, with them, the size of the arrow heads. Subgames can be added to the game tree with the syntax `\subgame«overlay»[<options>] at (<coord>) {<label>};` after all the branches of that specific subgame have been typeset. In order to get a visual separation between the nested subgames, we slightly reduce and increase, respectively, the `inner sep` of the inner and outer subgame from the default value of `3ex`. In order for β to lie fully within the subgame, we make its shape wider with the key `w=<factor>`. For `<factor>` equal to 0 and 1, respectively, the shape is a triangle and (a poor Bézier approximation of) half an ellipse. The default `<factor>` is 0.6. The depth of the subgame (how much it extends beyond the terminal nodes) is set so that subgames include the labels of terminal nodes in vertically oriented trees. Since those labels are not stored, we set the `depth` key manually for horizontal trees. To set it simultaneously for all subgames, we can either set the `subgame` key to `depth=<dimen>` at the environment level or we can set it globally with the command `\subgamesetoptions`. The `very sticky` key makes options `label=225` and `contour=delay` sticky until the end of the figure. If we used the `sticky` key instead, those options would remain active only until the next change of the active player through the `player` key. Note that `labels=short` and `force math` are still active since they were set globally by `\treesetoptions`.

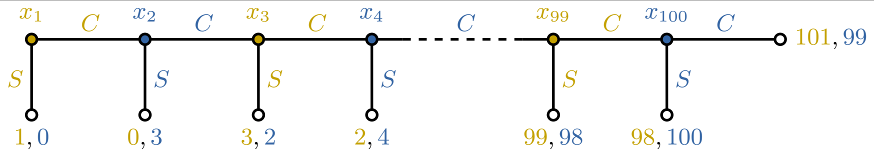


Fig. 17

```
1 \begin{gametree}[index=action, xscale=1.5, node autolabel=$x_{\enum}$]
2   \branch[root=w, player=1, offset={0.5,0}, sticky] from (0,0) to[v={1,0},h=1] {S(1,0); C};
3   \branch[player=2] from (wC) to {S(0, 3); C};
4   \branch[player=1] from (wCC) to {S(3, 2); C};
5   \branch[player=2, offset={0.8,0}] from (wCCC) to[h=1.6] {S(2, 4); C[etc]};
6   \branch[player=1, offset={0.5,0}, label text=$x_{99}$] from (wCCCC) to[h=1] {S(99,98);C};
7   \branch[player=2, label text=$x_{100}$] from (wCCCCC) to {S(98, 100); C[0](101, 99)};
8 \end{gametree}
```

Even though this game tree looks as if it was `horizontal`, it is easier to typeset in `vertical` orientation so that payoffs are placed below the terminal nodes by default. The different vertical distance of the children is again handled by providing a comma-separated list to the `v` key and correcting for horizontal alignment with the `offset` key. Specifically, each child is shifted from its default placement by `offset={0.5,0}`. Because we need the same `offset` for most branches, we make it `sticky`. By providing the `etc` option to a child, the center half of the branch is dashed to indicate a repeating pattern. Finally, the terminal payoffs of the last node are set to the right because the `0` in `C[0]` specifies the angle at which the terminal payoffs are set. Note that this option is overloaded: only numbers outside $(0, 1)$ are interpreted as the angle of terminal payoffs, whereas a number within $(0, 1)$ is used to specify the position of the edge label; see Figure 15.

Nodes are scale and shear invariant: they are displayed as circles of the same size even though the `gametree` environment is stretched by a factor of 1.5 in the x-direction by the `xscale` key. Any environment can be stretched in the y-direction with the yscale key. The `node autolabel` key provides an automated way of consecutively labeling nodes, where `\enum` is replaced by a counter that is increased by 1 for each node. A similar effect can be achieved by passing the `autolabel` key to a `\branch`, in which case automated labeling starts with the parent of that branch. Auto-labeling supports player-specific counters; see Figure 19 below for an example. The auto-labeling is overridden by providing a `label text` explicitly as on lines 6 and 7 of the code.

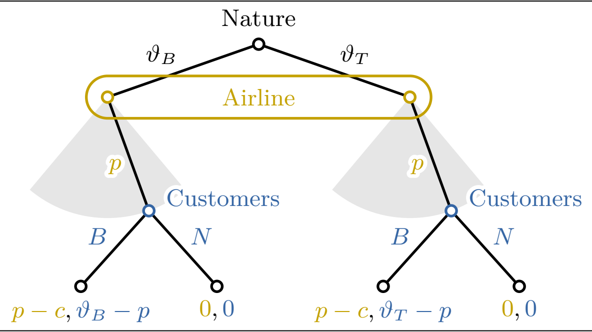
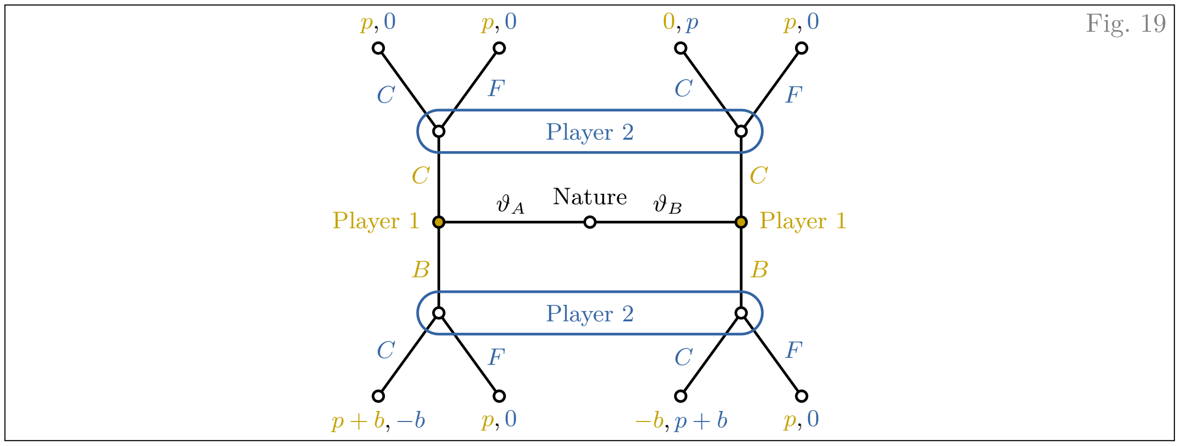


Fig. 18

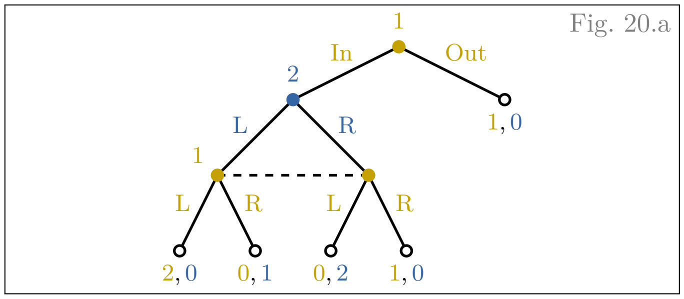
```
\begin{gametree}[player names={Airline, Customers}, nodes=hollow]
  \branch[root=w] from (0, 0) to[v=0.7, h=4] {\vartheta_B; \vartheta_T};
  \foreach[count=\i] \t in {B, T} {
    \branch[player=1,label=none] from (w\i) to[arc=230:310:1.6] {290:p[r]};
    \branch[player=2,label=10] from (w\i1) to[v=1,h=1.8] {B(p-c,\vartheta_{\t}-p); N(0,0)};
  }
  \informset[player=1] (w1) -- (w2);
\end{gametree}
```

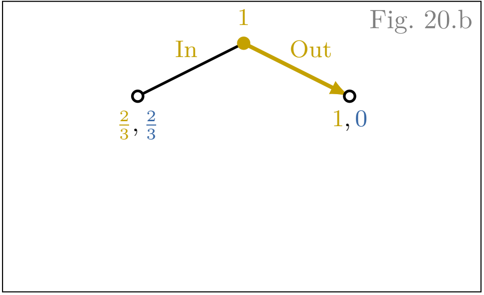
If `player names` are explicitly provided to the `gametree`, the key `labels=long` is set locally for this tree. A branch with a continuum of actions is typeset with `arc=<angle 1>:<angle 2>:<radius>` as in a regular Ti*k*Z arc. Individual actions are added to a continuum branch by preceding the action label with `<angle>:` or `<angle>:<radius>:` if the radius is to be different from the arc's. By default, an `arc` issues a contour for any action labels as well as the label of the next `\branch`'s parent node.



```
\resetplayernames
\begin{gametree}[labels=long, inform=base line]
  \branch[root=w] from (0,0) to[v=0,h=4] {\vartheta_A; \vartheta_B};
  \branch[player=1, horizontal] from (w1) to[v=2.4,h=0] {C; B};
  \branch[horizontal, label=right] from (w2) to {C[r]; B[l]};
  \branch[player=2, node=clear] from (w11) to[v=-1.1,h=1.6] {C[0.6](p, 0); F[0.68](p, 0)};
  \branch from (w21) to {C[0.68](0, p); F[0.6](p, 0)};
  \branch from (w12) to[v=1.1] {C[0.6](p+b, -b); F[0.68](p, 0)};
  \branch from (w22) to {C[0.68](-b, p+b); F[0.6](p, 0)};
  \foreach \i in {1, 2} {\informset (w1\i) -- (w2\i);}
\end{gametree}
```

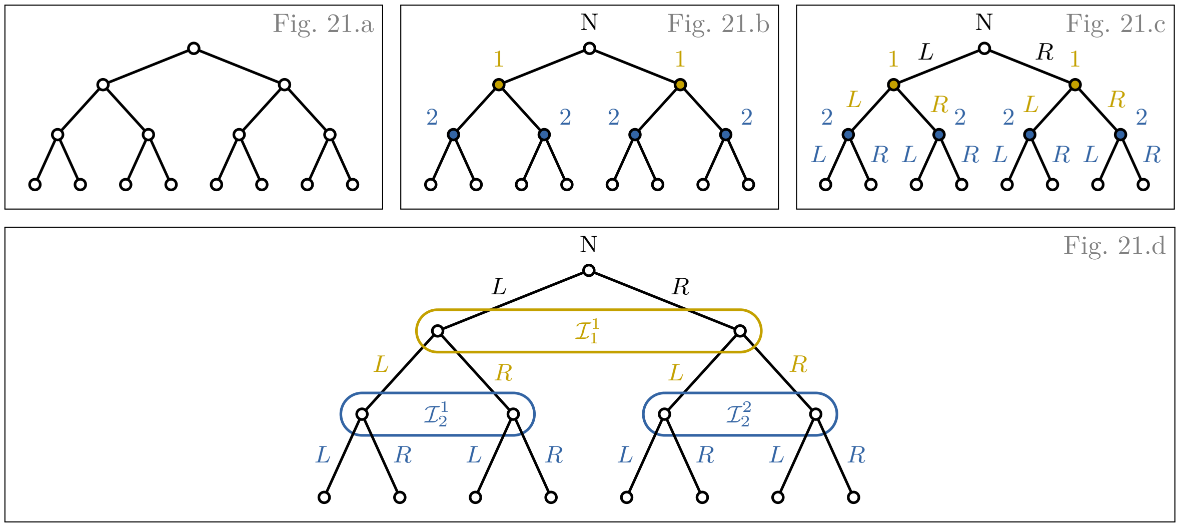
No game theory package documentation is complete without an example of a signaling game. By passing the `horizontal` key to individual branches, the orientation can be changed locally, allowing the tree to grow in arbitrary directions. Similar to the `label` key, the `node=clear` key remains active until the active player changes through an invocation of the `player` key. Because of the width of the information sets, we shift action labels closer to terminal payoffs with options `[0.6]` and `[0.68]`. The command `\resetplayernames` resets the player's names to Player 1, Player 2, etc. With the `inform` key, any keys can be passed to information sets within this game tree, such as the `base line` key here. The `base line` key aligns Player 2 properly within the information sets.





```
\begin{center}
\begin{gametree}[force math=false, nodes=full]
  \branch[player=1, root=w] from (0,0) to[v=0.7, h=2.8, pos=0.42] {In; Out<3->(1, 0)};
  \branch<1,4->[player=2, alt=({\frac{2}{3}, \frac{2}{3}})] from (w1) to[v=1, h=2] {L; R};
  \branch<1,4->[player=1, label=none, sticky] from (w11) to[h=1] {L(2, 0); R(0, 1)};
  \branch<1,4-> from (w12) to {L(0, 2); R(1, 0)};
  \informset<1,4->[dashed] (w11) -- (w12);
\end{gametree}
\end{center}
```

The `alt` key specifies that the branch starting at player 2's node is replaced with the terminal node $\frac{2}{3}, \frac{2}{3}$ on those overlays, on which the branch is not drawn. If the `alt` key is not specified, nothing is drawn. Shown above are overlays 1 and 3 produced by the code. The option `pos=0.42` sets the default edge label placement at 42% of the way from parent to children.

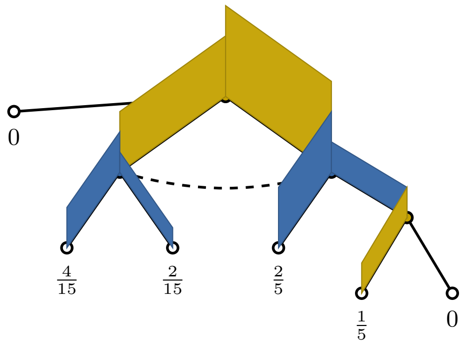


```
\begin{gametree}[node labels=<2->, edge labels=<3->, nodes={<1,4->clear}, inform
↪  autolabel=$\noexpand\mathcal{I}_{\pl}^{\enum}$]
  \branch[root=w, node=<2->] from (0,0) to[v=0.8, h=4] {L; R};
  \foreach \i in {1, 2} {
    \branch[player=1] from (w\i) to[v=1.1, h=2] {L[0.46+0.1*\i]; R[0.76-0.1*\i]};
    \informset<4->[player=2] (w\i1) -- (w\i2);
    \foreach \j in {1, 2} {
      \branch[label=240-100*\j] from (w\i\j) to[h=1, pos=0.62] {L(); R()};
  }}
  \informset<4->[player=1] (w1) -- (w2);
\end{gametree}
```

The game tree is gradually built up by providing overlay specifications to the environment keys `node labels`, `edge labels`, and `nodes`, as well as the `\informset` commands. Those specifications can be locally overridden: the key `node=<2->` in the root `\branch`, which is short for `node=<2->default`, ensures that the root is labeled even on overlays 4–, on which the key `nodes={<1,4->clear}` removes labels from all non-terminal nodes. Since key-value pairs are parsed as a comma-separated list, any commas within an overlay specification have to be hidden from the key-value parser by encapsulating the entire value with braces. Terminal nodes without payoffs are produced with an empty pair of parentheses `()`. Angles provided to the `label` key and positioning of edge labels are parsed with pgf's `\pgfmathparse` so any formulas are evaluated correctly.

With the key `inform autolabel=<code>`, information sets in the tree are labeled automatically according to the code provided. The `\pl` key is locally replaced with the active player's number and `\enum` is replaced by a counter that is increased for each information set. If `<code>` contains the macro `\pl`, player-specific counters are used in `\enum`, whereas a global counter is used if `\pl` does not appear in `<code>`. Automated labeling requires that `<code>` is expanded once so `\pl` and `\enum` are replaced with the correct values. This necessitates that any unexpandable macros (such as `\mathcal` here) are protected with a `\noexpand`; see also Footnote 3. A similar effect can be achieved by passing the `autolabel` key to an `\informset`, in which case automated labeling starts with that information set.

Fig. 22



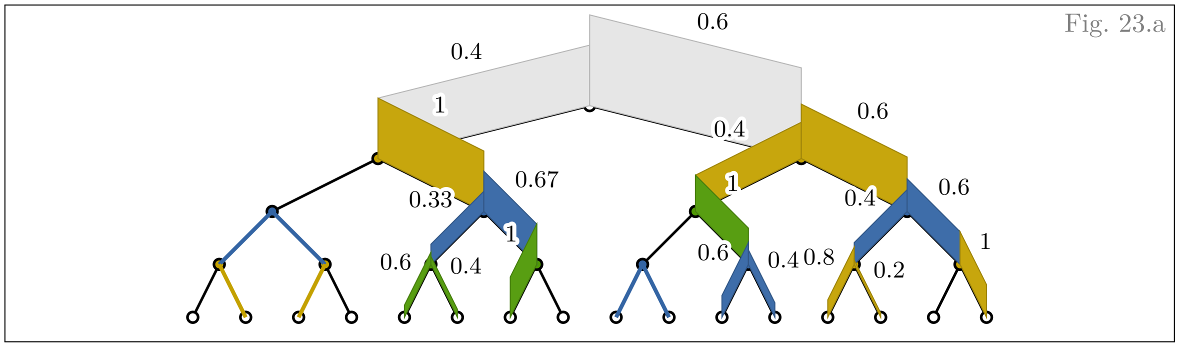
```
\begin{gametree}[labels=none]
  \branch[player=1, root=w] from (0, 0) to[v=-0.2, h=-2.8] {L()};
  \branch from (w) to[v=1, h=2.8] {M; R};
  \branch[player=2] from (w2) to[h=1.4] {\ell(); r()};
  \branch[offset={0.15,0}] from (w3) to[v={1,0.6}, h=1.7] {\ell(); r};
  \informset[curved=15] (w2) -- (w3);
  \branch[player=1] from (w32) to[v=1, h=1.2] {\lambda(); \rho()};
  \strategy<2->[prob=2, show terminal=<3->frac]{0, 0.4; 1 & 2/3}
\end{gametree}
```

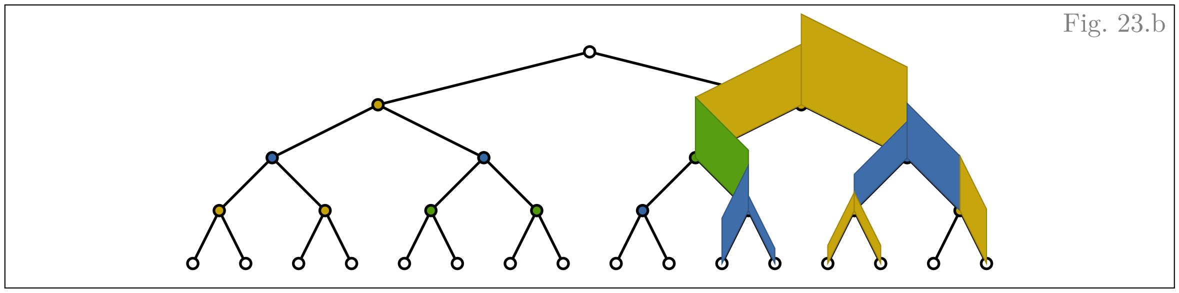
The probability distribution induced by a strategy profile can be visualized with the `\strategy` command. In its input, the players' strategies are separated by an ampersand, each player's information sets are separated by a semicolon, and the probabilities of each pure action within the information sets are separated by commas. The order of information sets and pure actions has to coincide with the order in which they were defined in the `\branch` commands.[7] The probability of the last pure action within each information set is not provided as an input since it is redundant. Any mathematical expressions that can be parsed by `\pgfmathparse` (such as fractions) are allowable as input. The `\strategy` command also allows input ordered by information sets rather than ordered by players. The above strategy profile could have been parametrized by `\strategy{0, 0.4; 2/3; 1}` as well.

The key `prob=<float>` indicates that one unit of probability is displayed with vertical height `<float>`. The default value is `prob=0`, for which the result looks as if the edges were highlighted directly with an overlay command. The key `show terminal` displays the resulting probabilities of each terminal node. Its optional values indicate an overlay specification for the terminal probabilities

[7]For non-singleton information sets, it is relevant at which point the first node in the information set was drawn, regardless of when the `\informset` command was issued.

as well as the option `frac`, which converts the output into fractions. One caveat to the `\strategy` command is that, currently, it is not compatible with the `gametree` option `index=action`.

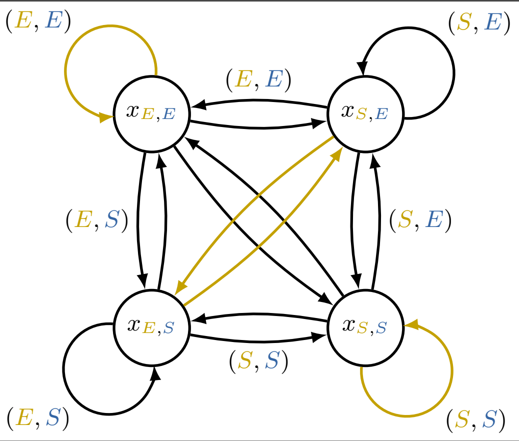




```
\begin{gametree}[labels=none]
  \branch[root=w] from (0, 0) to[v=0.7, h=5.6] {A; B};
  \foreach \i in {1, 2} {\branch[player=1] from (w\i) to[h=2.8] {A; B};}
  \foreach \i in {11, 12, 22} {\branch[player=2] from (w\i) to[h=1.4] {A; B};}
  \branch[player=3] from (w21) to[h=1.4] {A; B};
  \foreach \i in {111, 112, 221, 222} {\branch[player=1] from (w\i) to[h=0.7] {A(); B()};}
  \foreach \i in {211, 212} {\branch[player=2] from (w\i) to[h=0.7] {A(); B()};}
  \foreach \i in {121, 122} {\branch[player=3] from (w\i) to[h=0.7] {A(); B()};}
  \strategy<1>[prob=2, show prob=path]{0.4 & 0; 0.4; 0; 1; 0.8; 0 & 0.5; 1/3; 0.4; 0.2; 0.6
  ↪  & 0; 0.6; 1}
  \strategy<2->[prob=2, positive only, root=w2]{0.4 & 0; 0.4; 0; 1; 0.5; 0 & 0.5; 1/3; 0.4;
  ↪  0.8; 3/4 & 0; 0.6; 1}
\end{gametree}
```

The above two figures show both overlays. On the first overlay, the probability of each edge is displayed with the `show prob` key. Its value `path` causes the probabilities to be shown only on the path of the strategy profile. On the second overlay, the `root=<name>` key draws the conditional distribution, given that coordinate `(<name>)` is reached. By default, branches with probability 0 are still highlighted using the players' colors. This behavior can be turned off with the key `positive only`.

3.5 Automata



```
\begin{automaton}[ap, bend right=10, radius=5mm]
  \foreach[count=\i] \x in {{E, E}, {E, S}, {S, S}, {S, E}} {
    \node[atm] (x\i) at (90*\i+45:1) {$x_{\apm{\x}}$};
    \edge[path={mod(\i,2)}, loop=90*\i+45] (x\i) ->{\x};
  }
  \foreach[count=\i] \x in {{E, E}, {E, S}, {S, S}, {S, E}} {
    \pgfmathtruncatemacro{\j}{Mod(\i-2,4)+1}
    \edge (x\i) {\x}<-> (x\j);
  }
  \edge (x1) <-> (x3);
  \edge[path] (x2) <-> (x4);
\end{automaton}
```

Ti*k*Z was designed to draw graphs like automaton representations of strategy profiles: a series of nodes that are connected by arrows. In that sense, the basic functionality of Ti*k*Z already serves the purpose well; especially with the `automata` library provided by Ti*k*Z. Nevertheless, there is some redundancy that I aimed to remove with the `\edge` command.[8]

The package provides the `automaton` environment in order to set keys easily for the entire automaton. Automaton states are drawn as `\node`s with the `atm` style. Arrows between states are declared by the syntax `\edge <sequence of segments>;`. Each segment in the sequence is of the form `(<from coord>) <direction and labels> (<to coord>)`, where `(<from coord>)` and `(<to coord>)` have to be named coordinates and `<direction and labels>` is either `->`, `<-`, or `<->` with optional labels on either side of the arrows. If more than one segment is drawn, the `(<to coord>)` from one segment is used as the `(<from coord>)` of the next segment, i.e., `\edge (x1) -> (x2) -> (x3);` will draw arrows from `(x1)` to `(x2)` and from `(x2)` to `(x3)`. Arrows returning to the same state are drawn with the `loop=<angle>` key, where `<angle>` indicates the direction, in which the looping arrow is drawn.

The environment key `ap` encloses each `<arrow label>` with `\ensuremath{\ap{<arrow label>}}`. The key `bend right=<curv>` determines the direction and the curvature of the arrows. Arrows with the `loop` key are not affected by the curvature, but they are affected by the direction. The key `radius=<length>` sets the radius of each node to a minimum of `<length>`. Finally, by providing the `path` key to the `\edge` command, the arrows are highlighted.

[8]In automata, labels are always placed midway along the edge. With the standard Ti*k*Z notation, one would have to rewrite the statement `node[midway, <direction>]` for every arrow label. Moreover, with the standard Ti*k*Z notation, bidirectional arrows require two separate drawing instructions, which may complicate the code in some situations.

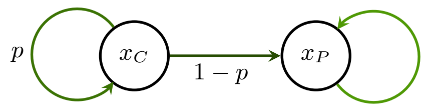


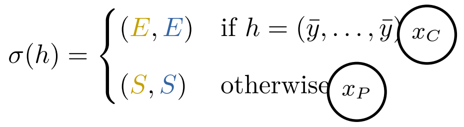
Fig. 25

```
\begin{automaton}[force math, straight, arrow=-stealth, alert color=player3]
  \foreach[count=\i] \x in {C, P} {\node[atm] (x\x) at (180*\i:1.2) {$x_\x$};}
  \edge[loop=left, path=0.75] (xC) ->{p};
  \edge[path=0.5, pos=0.46] (xC) ->{1-p} (xP);
  \edge[loop=right, path] (xP) ->;
\end{automaton}
```

As usual, the key `force math` places all edge labels in math mode. The key `straight` is shorthand for `bend right=0` and makes all non-looping arrows straight. The arrow shape can be set with the `arrow` key, where any values are acceptable that are understood by Ti*k*Z. Finally, the key `<highlight color>` sets the color, in which edges with the `path` key are highlighted in.

The key `path=<ratio>` mixes the `highlight color` with the package foreground color `xg-fg` according to the provided `<ratio>` between 0 and 1. In fact, this is how only the looping arrows for odd `\i` were colored in Figure 24. If the `<ratio>` is omitted, it is set to 1. To display the edge label at a different position than the middle along the path, we set the key `pos=<ratio>`.

Without any individual path options such as `path` and `pos`, the arrows could have been drawn also with the instruction `\edge (xC) -> (xC) -> (xP) -> (xP);`. Loops can be incorporated in such a sequence of edges by simply repeating the same coordinate. Since you cannot provide individual path options in this way, you will have to leave it to the algorithm to decide the direction of each loop.



$\sigma(h) = \begin{cases} (E, E) & \text{if } h = (\bar{y}, \ldots, \bar{y}) \\ (S, S) & \text{otherwise} \end{cases}$
x_C
x_P

Fig. 26

```
\[ \sigma(h) = \begin{cases}
    \ap{E, E} & \text{if }h=(\bar{y},\ldots,\bar{y}),\atmstate<2->[xshift=-1mm]{x_C}\\[2mm]
    \ap{S, S} & \text{otherwise}.\atmstate<2->{x_P}
  \end{cases} \]
```

To visualize which automaton states correspond to which instructions in the strategy profile, you can mark up the strategy profile with the command `\atmstate«overlay»[<options>]{<label}`. Note that `\atmstate` does not need to appear within a Ti*k*Z environment.

The somewhat overlapping design is intentional to give it the appearance as if the automaton states were thrown on top of the strategy profile in overlay 2. How much overlap is visually acceptable however, depends heavily on the content. The key `xshift=<dimen>` allows you to correct for that. Similarly, vertical alignment can be fine-tuned with the key `yshift=<dimen>`.

4 Usage

4.1 Package Dependencies

This package uses the `contour`, `listofitems`, `tikz`, `xcolor`, and the `xfp` package, all available on CTAN, as well as a small utility package `fikz` that is available on my website. If the package is loaded with the `external` option, the package `environ` is loaded additionally.

4.2 Package Options

The package comes with only relatively few package options. Most of the customization comes from the key system; see below. The package options are:

`ap` limits the size of the parentheses of `\ap` in textstyle math mode to `\big`. This will often look better if you use action labels with `\hat`s or `\tilde`s.

`contrast` disables coloring of all text commands and `matrixgame`s. Other environments are still in color. This option is intended for improving readability without disabling colors completely. For printing on black and white printer, use `greyscale` instead.

`draft` enables the contrast and the `nonode` option and it disables overlays and contours. All serves to obtain a minor improvement in compilation speed.

`external` enables support for Ti*k*Z's externalization library. Write `\externalize{<name>}` before any `tikzpicture` or any environment from this package to export the figures to a separate pdf. The name of the document is `<name>-<overlay>.pdf` in any non-handout beamer mode or `<name>.pdf` in beamer's handout mode or in any other document class. For the environments from this package, you can alternatively set the key `external=<name>` at the environment level. Externalization is, of course, very slow during the first compilation. However, figures will not have to be redrawn in any future compilations (unless there are changes to the figures). Make sure no two figures share the same `<name>`. *If you specify the same `<name>` for two or more figures, the figures are exported in every compilation because the metadata will have changed from the last compilation by the other figure sharing the same `<name>`.*

`greyscale` sets the `contrast` option and changes the player colors to grey scale.

`metropolis` sets the package's default foreground and background colors `xg-fg` and `xg-bg`, respectively, to the corresponding colors of the beautiful `metropolis` beamer theme.

`nodraft` ensures that the `draft` option is disabled even if the document class is loaded with it.

`nonode` improves compilation speed of game trees by drawing nodes as a circle-shaped path instead of a full-fledged Ti*k*Z node. As a consequence, Ti*k*Z paths ending in such a node will not be shortened, but rather end behind the node's center. The output of all commands from this package will look identical with or without the `nonode` option, but if you intend to mark up the trees with arrows, for example, you may wish to load the package without this option. This documentation, in fact, has been typeset with the `nonode` option.

4.3 Key System

The appearance of payoff matrices, belief spaces, game trees, and automata are governed by keys. Keys set at the level of an individual command are set only locally for that specific command. Keys passed to individual commands can be made active until the end of the environment by passing the `sticky` key. For `\branch` and `\informset`, sticky keys are active only until the active player changes (through an invocation of the `player` key). For these two commands, `very sticky` makes keys active until the end of the environment. Keys set at the level of an environment are set as the default for the entire environment, but can be overridden by keys at the command level. To change the default behavior globally, use the commands `\matrixsetoptions` for normal-form games, `\payoffsetoptions`, `\statesetoptions`, and `\informsetoptions` for payoff sets and belief spaces,

`\treesetoptions`, `\edgesetoptions`, `\subgamesetoptions`, and `\strategysetoptions` for game trees and `\atmsetoptions` for automata. Note that `\edgesetoptions` customizes edges in a game tree, not the `\edge` command in an automaton. Those are governed by `\atmsetoptions`. All keys that are available at the environment level are also available globally, but keys available for individual commands need not be available at the environment level and vice versa. Which keys are available at which level is indicated below. If the scope of a key differs from the standard behavior, it will be mentioned specifically.

4.4 Coloring Text

The basic idea is that each player has a dedicated color, a number, a name, and an "initial". The package supports at most nine players with numbers $1, \ldots 9$. Everything related to player i is typeset in that player's dedicated color. Player i's name can be accessed by the `\pli` command. Either player i's name or player i's number can be used to label payoff matrices, information sets, and nodes in game trees. Player i's initial or number can be used to label decision variables of that player. Player 0 is reserved for Nature, which follows special coloring rules in the `gametree` environment.

`\ap [<sep>]{<list of strings>}` is used to typeset action profiles or payoff vectors in the players' colors. It takes a comma-separated `<list of strings>` and typesets a `<sep>`-separated list of strings, delimited on both sides by parentheses, in which element i of `<list of strings>` is typeset in player i's color. The parentheses adjust in size to the height of `<list of strings>`, capped at `\big` in `textstyle` math if the `ap` package option is set.

`\apm [<sep>]{<list of strings>}` is used to typeset action profiles or payoff vectors within payoff matrices (hence the `m` in the macro name). It behaves like `\ap[<sep>]{<list of strings>}`, except that the colorized list of strings is not delimited by parentheses.

`\aps [<sep>]{<list of strings>}` is used to typeset a tuple (state, action profile), where the state is typeset in nature's color. The macro behaves like `\ap[<sep>]{<list of strings>}`, except that element i of `<list of strings>` is typeset in player $i - 1$'s color.

`\apsm [<sep>]{<list of strings>}` is the combination of `\apm` and `\aps`.

`\BR [<string>]` is used to denote the players' best-response correspondences. It typesets `\mathcal{B}_{<string>}` in player `<string>`'s color if `<string>` is a positive integer and it typesets `\mathcal{B}_{<string>}` without changing color otherwise.

Example: `\BR` and `\BR[i]` typeset $\mathcal{B}$ and $\mathcal{B}_i$ without changing the text color.

`\definevar [<options>]{<list of player variables>}` is used to make the players' decision variables easily accessible in the proper color. The argument `<list of player variables>` is a nested list of variables: the first layer separates players by semicolons and the second layer separates each player's variables by commas. If the variable is a `<string>`, a new macro `\<string>` is defined that typesets `<string>` in player i's color. If player i's variable is a `\macro`, the macro `\macro` is redefined to typeset the original content of `\macro` in player i's color. With the notation `\macro-><content>`, the macro `\macro` is (re)defined to typeset `<content>` in player i's color. If `\<string>` or `\macro` override existing macros, those are available again after the decision variable is released with `\releasevar`.

If `\<string>` or `\macro` is followed by a subscript, the subscript is typeset in player i's color as well. By setting the key `postfix=^` or `postfix='`, superscripts and primes are typeset in player i's color as well. With the key `prefix=<command name>`, an additional macro `\<string><command name>` or `\marco<command name>` is defined, which expands to `\<command name>{<string>}` or `\<command name>{\macro}`, respectively. When the key `player=<num>` is set, the variables before the first semicolon are treated as player`<num>`'s variables and the player counter increases with each semicolon.

Example: The command `\definevar[player=2, prefix=tilde]{\alpha; \beta}` defines the commands `\alpha` and `\alphatilde`, which typeset α and $\tilde{\alpha}$ in player 2's color, and the commands `\beta` and `\betatilde`, which typeset β and $\tilde{\beta}$ in player 3's color.

`\disablecolor` Saves player colors and sets them equal to the package foreground color `xg-fg`.

`\enablecolor` Restores player colors to the colors before `\disablecolor`.

`\letplayercolors` `[<first>]{<list of xcolors>}` sets the players' colors starting from player `<first>` to those specified in the comma-separated `<list of xcolors>`, where each color in the list must be understood as a color by the `xcolor` package. Specifically, `\letplayercolors` calls `\colorlet{playeri}{<xcolor i>}` from the `xcolor` package for each element `<xcolor i>` in `<list of xcolors>`. Use `\colorlet` directly for changing a single player's color.

`\multivar` `[<options>]{<list of variables>}` makes the players' decision variables easily accessible in the proper color if all the player's decision variable are denoted by the same letter. The argument `<list of variables>` is a comma-separated list of strings or macros. For each string variable `<string>`, a single-argument macro `\<string>{<num>}` is defined that typesets `<string>` in player `<num>`'s color. For each macro variable `\macro`, the macro `\macro` is redefined to a single-argument macro `\macro{<num>}` that typesets the original content of `\macro` in player `<num>`'s color. With the notation `\macro-><content>`, the macro `\macro` is (re)defined to a single-argument macro `\macro{<num>}` that typesets `<content>` in player `<num>`'s color.

The `index` key governs whether the variables are indexed by a subscript corresponding to player i. The default is `index=number`, which indexes the variable with player i's number. Option `index=players only` does the same, but suppresses coloring and indexing for `<num>=0`. Key-value pair `index=initial` places the initial letter of the players' names in the subscript and suppresses coloring and indexing for `<num>=0`. For `index=none`, no index is placed. For this option only, any following subscripts are typeset in player i's color. If the value is invalid, it is set to `number`. The keys `prefix` and `postfix` are available as they are for the `\definevar` command. The only other available key is `sticky`, which saves the current key-value pair as default for future calls of `\multivar`.

`\pl` `{<num>}{<string>}` for `<num>`$= 1, \ldots, 9$ typesets `<text>` in player `<num>`'s color. The argument may be omitted, in which case player `<num>`'s name is typeset. If a non-default `<text>` is to be typeset, `<text>` has to be enclosed by braces even if it is a single token.

`\plset` `{<num>}{<list of strings>}` for `<num>`$= 1, \ldots, 9$ typesets the comma-separated `<list of strings>` in player `<num>`'s color in textstyle math mode and encloses it in braces.

`\pltext` `{<num>}{<string>}` for `<num>`$= 1, \ldots, 9$ typesets `<string>` in player `<num>`'s color in text mode. This is a shorthand for `\text{\pl{<num>}{<string>}}`.

`\releasevar` `{<list of variables>}` releases variables defined by `\definevar` and `\multivar` and restores any macros that were overriden in the interim. The argument `<list of variables>`

is a comma-separated list of strings or macro names. The argument is optional and, if it is omitted, all variables defined by `\definevar` and `\multivar` are released.

It is best to avoid using `\definevar`, `\multivar`, and `\releasevar` within environments whose procedural code is executed repeatedly because it may lead to undefined macros. Examples of such environments are equations or frames if the beamer mode `second` is used.[9]

`\resetplayernames` Sets the player's names and initials to the default values "Player i" and "i", respectively.

`\setplayercolors` `[<first>]{<color model>}{<list of color specs>}` sets the players' colors starting from player `<first>` to those specified in the semicolon-separated `<list of color specs>`, interpreted in the `<color model>`. The `<color model>` and the color specification must be compatible with the `xcolor` package since that package's `\definecolor{playeri}{<color model>}{<color spec i>}` is called for each `<color spec i>` in `<list of color specs>`. Use `\definecolor` directly for changing a single player's color.

`\setplayername` `{<num>}{<name>}` sets player `<num>`'s name to `<name>` and player `<num>`'s initial to the first token of `<name>`. The defaults are reset by `\resetplayernames`.

`\setplayernames` `[<first>]{<list of names>}` sets the players' names starting from player `<first>` to the values in the comma-separated `<list of names>`. The default value of `<first>` is 1.

`\textset` `<num>{<list of strings>}` typesets the comma-separated `<list of strings>` in text mode and encloses it in braces. The `<strings>` are typeset in player `<num>`'s color if `<num>` is provided. The color is not changed if `<num>` is omitted. Note that there are no braces around `<num>` since LaTeX will understand `{<num>}` as the mandatory `{<list of strings>}`.

Example: `$\textset{1}{text}$` is typeset as {1}*text* since `{1}` is interpreted as the `{<list of strings>}` and `{text}` simply as a group that follows `\textset`.

4.5 Normal-Form Games

The dedicated environment `matrixgame` simplifies drawing of normal-form games by allowing you to specify keys for the entire game. However, all commands work in any `tikzpicture` environment. The following keys can be passed to the `matrixgame` environment as optional arguments:

`bottom = <list of strings>` typesets labels below the payoff matrices of this game by calling `\bottomlabel{<list of strings>}`; see the description below for details.

`br = <bool>` indicates whether the players' best responses in the payoff matrices will be computed and highlighted automatically with the `\br` command. The initial value is `false` and the default value (if the key is passed without value) is `true`. This option is supported only for games with at most three players.

`color = <color>` sets the color of the payoffs in the cells to `<color>`. If passed at the environment level, the default color of the labels are also set to `<color>`.

`debug = <bool>` indicates whether label placement is visualized for debugging purposes; see Figure 27. The initial value is `false` and the default value is `true`.

[9] As a specific example, suppose that `\A` is defined to $\mathcal{A}$ in the preamble and `\multivar{A}` is used in the beginning of an example to typeset A_i in player i's color. If you use `\releasevar{A}` within a frame that is executed twice, then the first call undefines `\A` and redefines it to $\mathcal{A}$, but the second call undefines `\A` completely.

```
\begin{matrixgame}[force math, debug]
  \payoffmatrix[h=0.7]{1, 3 & 4, -2\\ 0, -2 & 3, 1}
  \leftlabel{\alpha, \beta\varepsilon\tau\alpha}
  \toplabel{\lambda, \rho}
  \bottomlabel{$\theta$}
\end{matrixgame}
```

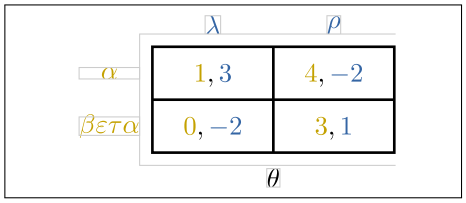


Figure 27: The thin border around the `\payoffmatrix` is set at a distance of one `label sep`. The intent of the `debug` key is to help you figure out why labels are not typeset the way you would like them to. For example, the option highlights that action labels are not baseline-aligned by default. You can change this by passing the `base line` option to `\leftlabel` and `\toplabel`, respectively. Moreover, the `\bottomlabel` is set at a distance of 1.4 `label sep`. You can set it closer to the matrix by passing the `tight` key to `\bottomlabel`.

`external = <name>` externalizes the resulting image as `<name>.pdf` or the resulting series of images as `<name>-<overlay>.pdf`, depending on the document class. This option requires the `external` package option, which enables Ti*k*Z's externalization library.

`font = <font spec>` adjusts the size or shape of the font used in payoff matrices, as well as the labels if the key is passed at the environment level. The initial value is `\normalsize`.

`force math = <option>` determines whether the contents of `\payoffmatrix` and the label commands are put into math mode. The options `matrix` and `label` set only one or the other in math mode; the options `true` and `false` set both inside or without math mode, respectively. The initial value is `matrix` and the default value (if the key is passed without value) is `true`.

`h = <num>` specifies the height of each cell in Ti*k*Z units. The initial value is `0.8`.

`immediate = <option>` determines whether `\payoffmatrix`, `\strike`, `\drawpayoffs`, and the label commands typeset their content immediately or whether they parse the input only and typeset the output at the end of the `matrixgame` environment. Admissible values of `<option>` are `true`, `false`, and `text`. If set to `text`, labels and payoff matrices are typeset immediately, but `\strike` and `\drawpayoffs` are not. If set to `false`, all commands are typeset at the end of the `matrixgame`. If set to `true`, all commands are typeset immediately. The option `immedate=true` may help marking up normal-form games, but `immediate=false` is necessary to pass labels as keys and to determine best responses of 3-player games with key `br`. Moreover, payoff matrices are parsed first, followed by label commands, which are then followed by `\strike`s and `\drawpayoffs`. In a regular `tikzpicture` environment, `immediate=true` is always set. The initial value is `text` and the default value (if the key is passed without value) is `true`.

`label font = <font spec>` adjusts the size or shape of the font used in labels. The initial value is `\normalsize`.

`label sep = <length>` specifies the distance, at which the labels are typeset from the payoff matrix. The `\toplabel`'s bounding box (baseline) is set at distance `<length>` if the key `vertical align=center` (`vertical align=baseline`) is set. The `\bottomlabel`'s bounding box is set at distance `1.1<length>` if it is a 3-player game or the option `tight` is passed to `\bottomlabel`. It is set at distance `1.4<length>` otherwise.

`left = <list of strings>` typesets labels to the left of the payoff matrices of this game by calling `\leftlabel{<list of strings>}`; see its description below for details.

```
\begin{matrixgame}[left={A, B}]
  \payoffmatrix{0 & 1 \\ 2 & -1}
  \toplabel[color=xg-fg]{H, L}
\end{matrixgame}
\begin{matrixgame}[top left={A, B}, player=1]
  \payoffmatrix{0 & 1 \\ 2 & -1}
\end{matrixgame}
```

Figure 28: The `color` and `player` keys allow alternative ways of writing the center and right-most matrix in Figure 3. Use commands `\leftlabel`, `\toplabel`, and `\bottomlabel` to pass options to labels. The keys `player=i` and `color=playeri` are mostly interchangeable except if passed at the environment level: the former colors only payoff matrices, whereas the latter also colors labels.

`left align = <option>` specifies whether `\leftlabel`s are `center`, `left`, or `right` aligned.

`line width = <length>` specifies the line width of the payoff matrix. You can also change the line width with the 7 default TikZ line width keys `thin`, `thick`, `very thick`, etc.

`list sep = <string>` sets the separator of label arguments to `<string>`. The initial value is `,`.

`name = <string>` assigns the name `<string>` to the parsed payoff matrix. The parsed payoff vectors can later be recalled and drawn by `\drawpayoffs[name=<string>]`.

`player = <num>` sets the color of the payoffs in the cells to those of player `<num>`.

`players = <bool>` determines whether the player's names are shown next to their actions. The initial value is `false` and the default value (if the key is passed without value) is `true`.

`player names = <list of strings>` sets the players' names starting from player 1 to the strings in `<list of strings>` and sets the key `players=true`. The player names are set globally, hence will be available after the `matrixgame` through the commands `\pli`.

`pos = {<x>,<y>}` places the center of the payoff matrix at Ti*k*Z coordinate (`<x>`, `<y>`).

`single player = <option>` specifies the default colors of labels and payoffs when payoffs of only a single player are specified. Admissible values for `<option>` are `default`, `zero-sum`, and `nature`. The former two make no changes to label colors and typeset payoffs in black and player i's color, respectively, where i is the player specified in the `zero-sum` key (set to player 1 initially). The value `nature` sets the color of payoffs and the top label to `xg-fg`.

`strike width = <length>` specifies the line width of strikes drawn with the `\strike` command.

`top = <list of strings>` typesets labels above the payoff matrices of this game by calling `\toplabel{<list of strings>}`; see its description below for details.

`top left = <list of strings>` typesets labels above and to the left of the payoff matrices by calling `\toplabel{<list of strings>}` and `\leftlabel{<list of strings>}`.

`vertical align = <option>` specifies how cell contents are vertically aligned, where `<option>` is either `default` (initial value), `baseline`, or `center`. The option `center` centers the text vertically, based on the text's total height (= height + depth). The option `baseline` aligns the vertical center above the baseline. The option `default` aims to reproduce the `baseline` option without measuring the labels for the sake of compilation time. This works correctly if labels and cell contents have the height of a capital A and the depth of a comma.

`w = <num>` specifies the width of each cell in Ti*k*Z units. The initial value is `1.6`.

`zero-sum = <num>` is short-hand for `single player=zero-sum` and `player=<num>`. The default value and the initial value are 1.

All keys can also be set globally by passing them to the `\matrixsetoptions` command. The commands used to typeset normal-form games are the following:

`\bottomlabel «overlay spec»[<options>]{<list of strings>}` places string i in the comma-separated `<list of strings>` below matrix i of this game. Admissible `<options>` are localized versions of the environment keys `color`, `list sep`, and `player`, as well as `base line=<bool>` and `tight=<bool>`. If the payoff vectors within the `\payoffmatrix` commands are at least 3-dimensional, the default color is player 3's color and the `tight` option is enabled by default; see `label sep` above. If the payoff vectors are at most 2-dimensional, the default color is `xg-fg` and `tight` is disabled by default.

`\br «overlay spec»{<text>}` underlines `<text>` on the slides indicated by `<overlay spec>` and displays `<text>` without underline on the remaining slides.

`\leftlabel «overlay spec»[<options>]{<list of strings>}` places string i in the comma-separated `<list of strings>` to the left of row i in each payoff matrix of this game. Admissible `<options>` are localized versions of the environment keys `color`, `list sep`, and `player`, as well as `base line=<bool>` and `align` with possible values `left`, `center`, and `right`. The default color is `player1`, the initial horizontal alignment is `center`, and the initial vertical alignment is set by the `vertical align` key at the environment level.

`\payoffmatrix «overlay spec»[<options>]{<table>}` draws a table with the payoffs indicated in `<table>`. In the argument `<table>`, rows are separated by `\\`, columns are separated by `&`, and payoffs of the different players are separated by commas. By default, the payoff vector within each cell is put into math mode, that is, the `force math` key is set to `matrix`. Thus, `$` signs are not needed in the input.[10] The environment keys `br`, `color`, `font`, `force math`, `h`, `line width`, `player`, `pos`, `vertical align`, and `w`, as well as the 7 default TikZ line width keys `thin`, `thick`, `very thick`, etc., can be passed to a `\payoffmatrix`.

`\strike «overlay spec»{<option>}{<num>}` is used to strike out the players' actions simultaneously in all payoff matrices of this game on the slides specified by `<overlay spec>`. The `<option>` can take the values `h` for striking out row `<num>` horizontally, `v` for striking out column `<num>` vertically, and `x` for striking out matrix `<num>`.

`\toplabel «overlay spec»[<options>]{<list of strings>}` places the string at position i in the comma-separated `<list of strings>` above column i of each payoff matrix. Admissible `<options>` are localized versions of the environment keys `color`, `list sep`, and `player`, as well as `base line=<bool>`. The default color is governed by the `single player` key and the initial alignment by the `vertical align` key set at the environment level.

4.6 Payoff Sets

Pictures of payoffs and payoff sets can be drawn by hand using the `\state` and the `\draw[edge]` commands, where edge is simply a Ti*k*Z style that inherits the line width from the states. Drawing

[10]This choice was made so that payoff vectors can be parsed easier for `\drawpayoffs`.

```
\begin{beliefspace}
  \state[autolabel=$\omega_{\enum}$] at (0, 0);
  \foreach \x in {-1, 0, 1} {\state at (\x, -0.9);}
\end{beliefspace}
```

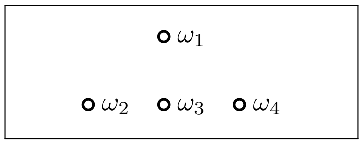


Figure 29: Equivalently, the key `state autolabel=$\omega_{\enum}$` could have been passed to the `beliefspace` environment. If states are placed in a rectangular left-aligned grid, use `\stategrid` instead.

payoff sets by hand is easier with the `nonode` package option since paths connecting the states will not be shortened. If the payoff vectors have been parsed by `\payoffmatrix`, the payoff vectors and their convex hull can be drawn automatically with the command `\drawpayoffs`.

`\state «overlay»[<options>] (<name>) at (<coord>) «label overlay»{<label>};` is used to place payoff vectors or nodes at coordinate `(<coord>)`. The only mandatory argument is `<coord>`. If `<name>` is provided, a Ti*k*Z node or coordinate with name `(<name>)` is placed at `(<coord>)`. The coordinate is placed on all slides of a frame, but it is visible only on those indicated by `<overlay>`. If the `<label>` is provided, it's center is placed in the direction indicated by the `label` key. Since nodes are only a visual representation of a dimensionless point, their size does not change when the image is scaled.

The available key-value pairs that can be passed as `<options>` to the `\state` command are the following. They can also be set globally by passing them to `\statesetoptions`.

`autolabel = <code>` automatically labels all following states in the `beliefspace`, starting with the current state. Use the macro `\enum` in the `<code>` as a placeholder for the value of a counter that is increased with each state; see Figure 29 for an example.

The `<code>` is fully expanded so that `\enum` is replaced with the correct value. Any non-expandable macros appearing in `<code>` must be protected with a `\noexpand`.

`base line = <bool>` vertically aligns the height above the label's baseline if set to `true`. Otherwise, the label is vertically centered based on its total height (= height + depth). The initial value is `true` in a `beliefspace` environment and `false` otherwise. The default value (if the key is passed without value) is `true`.

`color = <color>` sets `line color` and `label color` to `<color>`.

`contour = <option>` determines whether a contour of the package background color `xg-bg` is drawn around the label. The initial value is `false` and the default value (if the key is passed without value) is `true`. The contour is drawn using the `contour` package. Setting `contour=delay` is equivalent to setting `contour=true` and `delay=true`.

`debug = <option>` determines whether the placement of the label is visualized for debugging purposes; see Figure 27. Admissible values for `<option>` are `true`, `detail`, and `false`. The initial value is `false` and the default value (if the key is passed without value) is `true`.

`delay = <bool>` determines whether the state is drawn at the end of the environment. In either case, the coordinate is defined immediately. As with the `nonode` package option, path shortening is not available if `delay` is set to `true`.

```
\begin{tikzpicture}
  \state[label=60, debug=detail, sticky] at (0, 0) {Label};
  \state[label placement=box] at (2, 0) {Label};
\end{tikzpicture}
```

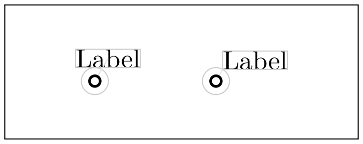


Figure 30: The `center` algorithm places the center of the label's bounding box at the angle specified by the `label` key such that the closest point is `label sep` away from the state. One advantage of this algorithm is that it is continuous in the specified angle. The `box` algorithm instead places the closest point of the label's bounding box at the angle specified by the `label` key. One advantage of this algorithm is that the label will never intersect a line going through the state perpendicular to the angle (as it is the case for edges in game trees).

`delay contour` = `<bool>` determines whether the `\state`'s `<label>` is typeset at the end of the environment whenever the `contour` key is active to place the contour in front of any other objects. As with the `delay label` key, `<label>` is expanded once.

`delay label` = `<bool>` determines whether the `\state`'s `<label>` is typeset at the end of the environment. The initial value is `false` and the default value (if the key is passed without value) is `true`.

`fill` = `<color>` determines the color of the state's interior. The default value is the active player's color. For player 0 (active initially), it is `xg-bg`, the package background color.

`font` = `<font spec>` adjusts the font of the label. The initial value is `\small`.

`label` = `<direction>` specifies the direction, in which the label is placed. Accepted values for `<direction>` are the eight cardinal directions `left`, `above left`, `above`, etc. as well as any number, interpreted as angle in degrees. The initial value is 0.

The eight cardinal directions `left`, `above left`, `above`, etc. are also defined as their own keys: `\state[below right] at (0, 0) {A};` places the label A below and to the right of the state. To replicate the behavior of standard Ti*k*Z nodes with these direction keys, the label placement algorithm is set to `box`. The result is thus slightly different than for `\state[label=-45] at (0, 0) {A};`, which keeps the current label placement algorithm.

`label color` = `<color>` determines the color of the label. The default color is the active player's color.

`label placement` = `<option>` specifies which label placement algorithm is used, where `<option>` may take the values `center`, `box`, or `simple`. For each algorithm, the distance from the anchor point to the label's bounding box is determined by the `label sep` key. The `center` algorithm places the center of the label in the specified direction of the anchor point. The `box` algorithm places the closest point of the bounding box in the specified direction of the anchor point; see Figure 30. The `simple` algorithm places the closest point of the bounding box in one of the eight directions `above`, `above right`, etc. of the state. Note that the latter two are discontinuous in the specified direction. The default algorithm is `center`.

`label sep` = `<length>` specifies the distance of the closest point of the label's bounding box from the outer rim of the circle representing the state. The initial value is `1ex`.

`line color` = `<color>` draws the state's outline in `<color>`. The default color is `xg-fg`.

`line width` = `<length>` specifies the line width used to draw the state. You can also change the line width with the 7 default TikZ line width keys `thin`, `thick`, `very thick`, etc. Changing the width will also change the state's radius so that the state's interior stays the same; see

```
\begin{tikzpicture}
  \foreach \i in {1, 2, 3} {\state[line width=\i pt] at (\i, 0);}
  \foreach \i in {2, 3, 4} {\state[radius=\i pt] at (\i-1, -0.7);}
\end{tikzpicture}
```

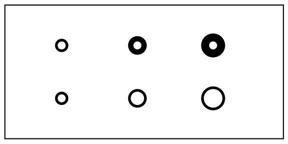


Figure 31: Increasing the `line width` also increases the state's radius.

Figure 31. Moreover, the line drawn by the Ti*k*Z style `edge` is set to the same width. The initial value is `0.8pt` in `beamer` and `1pt` in other document classes.

`radius = <length>` specifies the radius of the state. The initial value is `1.5pt` in `beamer` and `2pt` in other document classes.

An automated way to draw payoffs and their convex hull is provided by the `\drawpayoffs` command, provided that the payoffs have been parsed by `\payoffmatrix`. Needless to say that this gimmick is slower than when payoffs are drawn by hand. Thus, this feature may not have a place in long lecture slides that are compiled many times, but I do find it helpful when creating assignment questions to check at a quick glance which payoff vectors can be attained by, say, trigger strategies.

`\drawpayoffs «overlay spec»[<options>]` draws payoff vectors parsed by `\payoffmatrix`, their convex hull by means of the Graham scan, as well as coordinate axes.

The following are the keys that can be passed as `<options>` to the `\drawpayoffs` command or they can be set globally by passing them to `\payoffsetoptions`.

`axis = <factor>` sets the length of the drawn axes equal to `<factor>` times the difference between maximal and minimal payoffs parsed in either direction.

`force math = <bool>` indicates whether . The initial value is equal to `true` if `force math=matrix` or `force math=true` was specified in a `\payoffmatrix` in the same environment. Otherwise, the initial value is `false`. The default value (if the key is passed without value) is `true`.

`labels = <bool>` specifies whether the payoff vectors are labeled. If they are, the labels are placed facing away from the center of mass of all payoff vectors drawn. The initial value and the default value (if the key is passed without value) are both equal to `true`.

`line width = <length>` specifies the line width used to draw the payoffs and their convex hull. You can also change the line width with the 7 default TikZ line width keys `thin`, `thick`, `very thick`, etc. Changing the width will also change the states' radius so that the states' interior stays the same.

`matrix = <num>` indicates which payoff matrix of name `name` is to be drawn.

`name = <string>` causes the macro to draw the payoff vectors in matrix `matrix` stored by `\payoffmatrix` under the name `<string>`. Without specifying a name in either macro, the most recently read `\payoffmatrix` is drawn.

`scale = <factor>` scales the drawn payoffs by `<factor>`.

`shift = {<x>,<y>}` shifts the origin of the drawn payoffs to coordinate (`<x>`, `<y>`) in the underlying Ti*k*Z coordinate system.

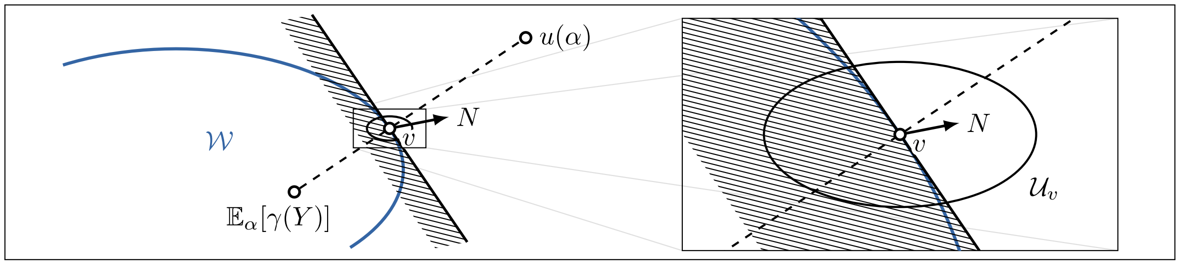


Figure 32: The normal vectors of `\halfspace`s are scale-invariant, but they are purposely designed not the be shear-invariant in order to accurately reflect coordinate transformations. If one redraws Figure 6 with environment options `[yscale=0.8, xscale=1.5]`, one obtains the above figure. Observe that `\state`s are both scale- and shear-invariant as they are just a visual representation of a dimensionless point.

`tikz nodes = <bool>` determines whether states are implemented as a full-fledged Ti*k*Z node with path-shortening features or only as a coordinate with a circle-shaped path representation. If the `nonode` package option is specified, the initial value is false, otherwise it is true. The default value (if the key is passed without value) is `true`.

`xscale = <factor>` scales the drawn payoffs by `<factor>` in the x-direction.

`yscale = <factor>` scales the drawn payoffs by `<factor>` in the y-direction.

Half spaces are drawn with the `\halfspace` command.

`\halfspace «overlay spec»[<options>] at (<coord>) dir (<normal>) {<label text>};` draws the lower half-space through `<coord>` in the direction of the normal vector `<normal>`. If a <label text> is provided, the normal vector is labeled by `<label text>`. The length of the normal vector is unaffected by scale. If `<normal>` consists only of an angle, rather than a full coordinate, the normal vector is suppressed.

All of its keys can be set at the macro level or globally with the command `\halfspacesetoptions`.

`contour = <bool>` determines whether `<label text>` is typeset with a contour of the package background color `xg-bg`. There is no `delay` option for half-spaces.

`depth = <length>` specifies the depth of the hatching pattern.

`hatch = <length>` specifies the distance between two consecutive lines of the hatching pattern. The hatching pattern is unaffected by changes to scale.

`l = <length>` specifies how far to the left of `<coord>` the half-space extends to. The initial value is half of the width specified in the `w` key.

`label = <direction>` determines the `<direction>`, at which `<label text>` is typeset from the tip of the normal vector. Accepted values for `<direction>` are the eight cardinal directions `left`, `above left`, `above`, etc. as well as any number, interpreted as angle in degrees. The initial value is the direction of `<normal>`.

The eight cardinal directions `left`, `above left`, `above`, etc. are also defined as their own keys. To replicate the behavior of standard Ti*k*Z nodes with these direction keys, the label placement algorithm is locally set to `box`.

`w = <length>` specifies the width of the `\halfspace`. The half space extends equally to both sides unless the `l` key is set.

A convenient way to zoom into any Ti*k*Z image is the `\zoom` command.

`\zoom «overlay spec»[<options>]{<figure>}<markup>` displays both the `<figure>` and a zoom-in of it, in which the zoomed-in part is marked up by `<markup>`. The zoom-in is shown only on the slides specified by `<overlay>`. The size of nodes as well as the length of the normal vector and the hatching pattern of half spaces are unaffected by the zoom.

The `\zoom` command is incompatible with any `delay` options. Named coordinates are locally redefined within the zoom-in, but reset to the original coordinates afterwards. Any delayed features are thus missing from the zoom-in. Because delayed features are not supported, the `nonode` package option is locally disabled within a `\zoom` command.

There are four mandatory keys to be specified with the `\zoom` command.

`l = {<x>,<y>}` specifies the lower left coordinate `(<x>,<y>)` of the zoomed-in area. `xg-bg`. There is no `delay` option for halfspaces.

`scale = <factor>` specifies the scale factor of the zoomed-in area.

`shift = {<x>,<y>}` shifts the center of the zoomed-in part by `(<x>,<y>)` in the non-zoomed-in coordinate system.

`u = {<x>,<y>}` specifies the upper right coordinate `(<x>,<y>)` of the zoomed-in area.

4.7 Belief Spaces

The main commands used to draw belief spaces are `\state` and `\informset`. The `\state` command is as described above, except that the `label=<direction>` key is ignored in a `beliefspace` environment. Any state's label is set to the right of the state to ensure that information sets also contain and pads the labels. The available options for the `beliefspace` environment are the following.

`arc = <angle>` sets the total angle covered of arcs drawn with the `exterior=arc` option to `2*<angle>`. The initial value is 22.5.

`base line = <bool>` sets `<bool>` as the initial value for the `base line` key of all `\state`s and `\informset` within the `beliefspace`. The initial and default values are both `true`.

`center = <option>` determines whether states and their labels are horizontally centered within `\informset`s. Admissible values for `<option>` is a `<bool>` or a `<length>`. If set to `true`, `\state`s are drawn only at the end of the environment to ensure that the labels of all `\state`s have been measured. In order for labels appearing in a for loop to be typeset correctly, use `\xforeach`, which expands labels once before they are stored for later use. If `<option>` is a `<length>`, the states are placed immediately under the assumption that `<length>` is the width of the widest `\state` label.

In either case, the `center` option causes the coordinate `(<name>)` placed with the `\state` command to be misaligned with the provided `(<coord>)`; see below.

```
\begin{beliefspace}[radius={4mm, 5mm}, center, debug]
  \state (w1) at (0, 0) {$\omega_1$};
  \state (w11) at (0, 1.5) {$\omega_{11}$};
  \foreach \i in {2, 3}{\coord (w1\i) at (120*\i-30:1.5);}
  \informset[player=1, label=0:2] (w11) {$\tau_1$};
  \informset (w12) .- (w1);
  \informset[player=2] (w11) -- (w1) -. (w13) .- (w1);
  \informlabel (above:2:1pt) at (w11) {$\ap{1, 0.2}$};
  \informlabel (20:7mm) at (w1) {$\ap{0, 0.2}$};
\end{beliefspace}
```

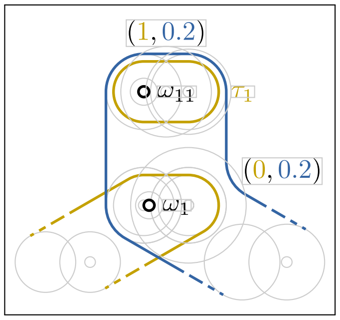


Figure 33: For `beliefspaces`, the `debug` key visualizes the left and right anchors, around which the information sets are drawn, where information sets are drawn to for half-open information sets, and how much the `center` key displaces each state. The key also visualizes that labels are anchored at the center for angles 90 and −90, but at the left and right anchors for any other angles. Moreover, we can see that labels of `\informsets` are baseline-aligned by default, whereas `\informlabels` are not. This example demonstrates that we can use the syntax `(w1) -. (w13) .- (w1)` to typeset non-convex open information sets. It also shows that label distances can be provided as either `<num>`, `<num>:<dimen>`, or `<dimen>`.

`contour = <option>` specifies the initial values of the `contour` key of all `\states` or `\informsets` within the `beliefspace`. Admissible values for `state` and `inform`, which set the initial value to `true` for `\state` and `\informset`, respectively, as well as `true` and `false`, which set the initial value to `true` and `false`, respectively, for both.

`debug = <option>` determines whether the placement of nodes, information sets, and labels are visualized for debugging purposes; see Figure 33. Admissible values for `<option>` are `true`, `detail`, and `false`. The initial value is `false` and the default value is `true`.

`delay contour = <bool>` specifies the initial values of the `delay contour` key of all `\states` or `\informsets` within the `beliefspace`. Admissible values for `state` and `inform`, which set the initial value to `true` for `\state` and `\informset`, respectively, as well as `true` and `false`, which set the initial value to `true` and `false`, respectively, for both.

`event = <length>` sets the default radius of an information set of player 0 to `<length>`.

`exterior = <option>` determines how information sets are drawn that contain disjointed sets of states. Allowable values for `<option>` are `arc` and `extend`; see Figures 12 and 34, respectively, for examples. The initial value is `arc`.

`external = <name>` externalizes the resulting image as `<name>.pdf` or the resulting series of images as `<name>-<overlay>.pdf`, depending on the document class. This option requires the `external` package option, which enables Ti*k*Z's externalization library.

`font = <font spec` adjusts the font of all labels in the belief space.

`identical width = <bool>` determines whether information sets are drawn as if all state labels had the same width to create a uniform look. The default and initial value are `true`.

Disabling this option is currently supported only if states are in a single row, as in Figure 9.

`inform autolabel = <code>` automatically labels all `\informsets` in the `beliefspace`. Use the macro `\pl` in the `<code>` as a placeholder for the player's number and the macro `\enum` as a placeholder for the value of a counter that is increased with each information set. If `<code>` contains the macro `\pl`, player-specific counters are used in `\enum`, whereas a global counter is used if `\pl` does not appear in `<code>`. See Figure 19 for an example.

The `<code>` is fully expanded so that `\pl` and `\enum` are replaced with the correct values. Any non-expandable macros appearing in `<code>` must be protected with a `\noexpand`.

`inform line width = <length>` specifies the default line width used to draw `\informset`s.

`left = <length>` adds `<length>` to the left of the measured width of the information sets, which causes the `\state` and its label to shift relative to its specified coordinate.

The `left` option causes the coordinate `(<name>)` placed with the `\state` command to be misaligned with the provided `(<coord>)`; see below.

`line width = <length>` specifies the default line width used to draw `\state`s and `\informset`s.

`radius = <list of lengths>` sets the default radiii of the various players' information set of the elements of the comma-separated `<list of lengths>`.

`right = <length>` adds `<length>` to the right of the measured width of the information sets.

`scale = <factor>` scales the belief space by `<factor>`.

`state = <bool>` specifies whether a state and label are drawn separately or whether the label itself represents the state. The initial value and the default value are both `true`.

`state autolabel = <code>` automatically labels all states in the `beliefspace` with `<code>`. Use the macro `\enum` in the `<code>` as a placeholder for the value of a counter that is increased with each state; see Figure 29 for an example.

The `<code>` is fully expanded so that `\enum` is replaced with the correct value. Any non-expandable macros appearing in `<code>` must be protected with a `\noexpand`.

`state inner sep = <length>` specifies how far from the label any markup will stop if the `state=false` option is set. See the pgf manual for a description of the `inner sep` key of a Ti*k*Z `\node`.

`state line width = <length>` specifies the default line width used to draw `\state`s.

`state radius = <length>` specifies the radius of `\state`s drawn in the belief space.

`tikz nodes = <bool>` determines whether states are implemented as a full-fledged Ti*k*Z node with path-shortening features or only as a coordinate with a circle-shaped path representation. If the `nonode` package option is specified, the initial value is false, otherwise it is true. The default value (if the key is passed without value) is `true`.

`xscale = <factor>` scales the belief space by `<factor>` in the x-direction.

`yscale = <factor>` scales the belief space by `<factor>` in the y-direction.

If the coordinate `(<name>)` defined with the `\state` command is misaligned with the specified `<coord>`, a coordinate `(<name>-is)` (for information set) is placed at `<coord>` and can be accessed if needed. If such misalignment takes place, the command `\informset (w1) -- (w2);` draws information sets around coordinates `(w1-is)` and `(w2-is)` instead of `(w1)` and `(w2)`; see Figures 9 and 33.[11] It is

[11]This is in most cases precisely what one wants if one specifies the `center` or `left` option, but it is good to be aware of it if one draws `\informset`s not just around `\state`s but also around regular Ti*k*Z `\coordinate`s.

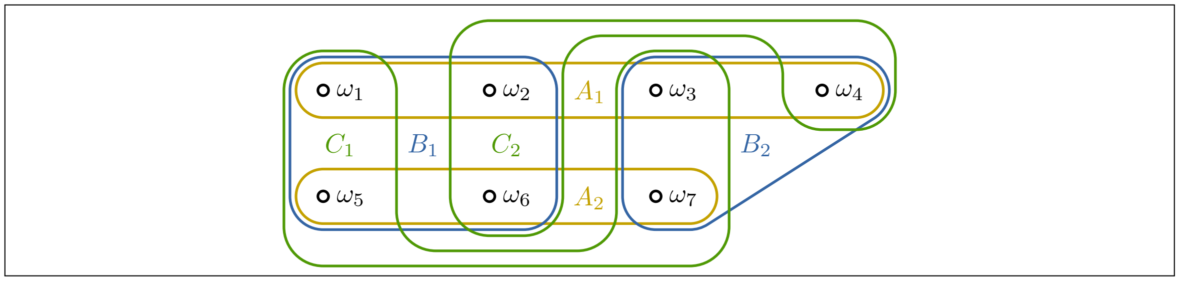


Figure 34: The code is identical to Figure 12 with two exceptions. First, the `exterior=extend` key is provided to the environment. Second, because information sets cannot have a non-convexity at the first node, line 11 is instead `\informset[player=3] (w1) -- (w5) ~~ (w7) -- (w3) -- (w7) ~~ (w5) {$C_1$};`

recommended to use the command `\coord` introduced below, rather than the Ti*k*Z native `\coordinate`, because `\coord` takes care of the horizontal misalignment if the `center` or `left` keys are provided.

States in a rectangular grid can be drawn with the `\stategrid` command.

`\stategrid` `«overlay spec»[<options>]<list of labels>;` is used to draw a rectangular grid of states. Unless the `autolabel` key is provided, rows in `<list of labels>` are separated by `\\` and columns are separated by `&`. If the `autolabel` is set, `<list of labels>` is instead a `\\`-separated list of integers specifying the number of states in each row.

The key-value pairs available for `<options>` are the following. Those keys can also be set globally by passing them to the `\gridsetoptions` command.

The `<code>` is fully expanded so that `\enum` is replaced with the correct value. Any non-expandable macros appearing in `<code>` must be protected with a `\noexpand`.

`autolabel = <code>` automatically labels the states in the grid with `<code>`. The macro `\enum` is used to refer to the states' indices, determined by the `enum` key.

`enum = <option>` specifies how states are indexed. Admissible values for `<option>` are `grid` and `line`. The former indexes a state through `<row><column>`, whereas the latter indexes states consecutively. The initial value is `line`.

`force math = <bool>` determines whether the labels of each state in the grid are put into math mode. The initial value is `false` and the default value (if the key is passed without value) is `true`.

`h = <num>` specifies the horizontal distance between two columns in Ti*k*Z units. The initial value is `1.2`.

`name = <name>` specifies the name of the coordinates that are assigned to the states in the grid. The coordinate of state at `<index>` is labeled `(<name><index>)`, where `<index>` is specified by the `enum` key. The initial value is `w`.

`v = <num>` specifies the vertical distance between two rows in Ti*k*Z units. The initial value is `1`.

Non-grid patterns of `\states` as in Figure 8 can be drawn efficiently with a for loop. In order for running variables to be expanded, use the `\xforeach` command.

`\xforeach [<options>] \macroname in {<list of items>}{<code to be repeated>}` behaves exactly like pgffor's `\foreach` command, except that the labels of any `\state`, `\informset`, `\informlabel`, and `\branch` appearing within the loop are expanded once. If any non-expandable macros appear in those labels, protect the frrom expansion by immediately preceding them with `\noexpand`; see also Figure 8 and Footnote 3s.

Aside from states, the main command used to draw belief spaces is the `\informset` command.

`\informset «overlay spec»[<options>] <list of coordinates> {<label text>};` is used to draw an information set that contains the states specified in the `<list of coordinates>` on the slides specified by `<overlay spec>`. The coordinates in `<list of coordinates>` are separated by `--` for default behavior, by `..` to indicate a repeating pattern with a dashed line, by `.-` and `-.` to draw a half-open information set in the direction of the dot, see Figure 9, or by `~~`, `~^~`, `~v~`, `~<~`, and `~>~` to draw information sets of a disconnected set of states, see Figures 12 and 34. The direction of the connection between different components is determined automatically for separator `~~`, and it can be provided manually by `~^~`, `~v~`, `~<~`, and `~>~` for connections above, below, to the left, and to the right, respectively. If one of the components consists of a single state, the direction of the connection must be provided manually. The environment key `exterior` governs how information sets with multiple connected components are drawn. The `<list of coordinates>` has to be provided in counterclockwise order and it has to start at a state on the convex hull (unless the `exterior=arc` key is set, in which case it can start at a state on the convex hull on any component). The `<list of coordinates>` is the only mandatory argument, the rest can be omitted. The `<label text>` overrides any default label assigned to the information set.

The key-value pairs available for `<options>` are the following. Those keys can also be set globally by passing them to the `\informsetoptions` command.

`autolabel = <code>` automatically labels all following `\informset`s in the `beliefspace`, starting with this information set. Use the macro `\pl` in the `<code>` as a placeholder for the player's number and the macro `\enum` as a placeholder for the value of a counter that is increased with each information set. If `<code>` contains the macro `\pl`, player-specific counters are used in `\enum`, whereas a global counter is used if `\pl` does not appear in `<code>`. See Figure 19 for an example.

The `<code>` is fully expanded so that `\pl` and `\enum` are replaced with the correct values. Any non-expandable macros appearing in `<code>` must be protected with a `\noexpand`.

`base line = <bool>` vertically aligns the height above the label's baseline if set to `true`. Otherwise, the label is vertically centered based on its total height (= height + depth). Both the initial value and the default value (if the key is passed without value) are.

`color = <color>` sets the color of the `\informset` and its label to `<color>`. The initial value is the active player's color, i.e. the player that was last invoked through the `player` key in either the `\state`, `\informset`, or `\branch` command.

`contour = <option>` determines whether a contour of the package background color `xg-bg` is drawn around the label. The initial value is `false` and the default value (if the key is passed without value) is `true`. The contour is drawn using the `contour` package. Setting `contour=delay` is equivalent to setting `contour=true` and `delay contour=true`.

`curved = <curv>` causes information sets to be drawn as curved dashed lines connecting the first and last state of the information set, where `<curv>` is the curvature of the line. The feature is initially turned off and the default value (if the key is passed without value) is `20`.

`dashed = <bool>` represents information sets as straight dashed lines connecting its states if set to `true`. This differs from `curved=0`, which connects only the first and the last state. The initial value is `false` and the default value (if the key is passed without value) is `true`.

`delay contour = <bool>` indicates whether the `\informset`'s `<label>` is typeset at the end of the environment whenever the `contour` key is active to place the contour in front of any other objects. The same conditions on expandability of the label apply as for the `\state`'s key. The initial value is `false` and the default value (if the key is passed without value) is `true`.

`font = <font spec>` adjusts the font of the label.

`label = <option>` determines how the label is placed. Admissible values for `<option>` are `none`, which suppresses any labels, `center`, which places the label at the barycenter of the information set, `square`, which places the label at the center of the square that contains the information set, a number `<float>` in $(0, 1)$, which places the label at a fraction `<float>` of the way from the first state to the last state in the information set, a `<direction>`, which places the label at angle `<degree>` from the first node of the information set, or a pair `<direction>:<label radius>`, which also places the label in the `<direction>` but at a potentially different radius. Admissible values for `<direction>` are any of the eight cardinal directions `left`, `above left`, `above`, etc. or an angle in degrees outside of the interval $(0, 1)$. Within a `gametree`, the initial value is `135` if the information set is `dashed` or `curved` and the initial value is `center` otherwise. In any other environment, the initial value is `none` if no `<label text>` is provided. If a `<label text>` is provided, the initial value is `center` unless the information set consists of several connected components, in which case it is `square` in the first connected component.

`label radius = <num>:<length>` indicates that the label is placed at a distance of player `<num>`'s radius plus `<length>` (plus `<label sep>`) from the center of the information set. It is also possible to specify only a player number `<num>` or only a `<length>`; see Figure 33.

`label sep = <length>` specifies the distance of the closest point of the label's bounding box from the outer rim of the information set.

`line = <line spec>` specifies how the line of an information set is drawn. Allowable values of `<line spec>` are any key-value pair that is understood by the Ti*k*Z draw command. The intended use is for values like `dashed`, `dotted`, etc.

`line width = <length>` specifies the line width used to draw the information set. You can also change the line width with the 7 default TikZ line width keys `thin`, `thick`, `very thick`, etc.

`player = <num>` sets the active player to `<num>`. This sets the default color and label to the active player's color and name, respectively, or the active player's number if the key `labels=short` is set. The change of active player is global, hence future calls of `\state`, `\informset`, and `\branch` use the same defaults unless the active player is changed.

`radius = <length>` specifies the radius of the information set.

Finally, the following auxiliary commands may be helpful when drawing hybrid figures, i.e., when information sets are drawn outside of the `beliefspace` environment.

`\coord (<name)> at (<coord>);` defines a coordinate at `(<coord>)`, whose name is `(<name>-is)` if the `left` or `center` keys are set and its name is `(<name>)` otherwise.

`\event «overlay»[<options>] <list of coordinates> {<label text>};` is a shorthand for an `\informset[player=0, <options>]` command with identical arguments.

`\informsetradius [<num>]{<list of lengths>}` sets the default radii, starting with player `<num>`, to the elements of the comma-separated `<list of lengths>`. The default value of `<num>` is `1`.

Example: The command `\informsetradius[0]{5mm, 3mm, 4mm}` sets the environment keys `event=5mm` and `radius={3mm, 4mm}` globally.

`\informlabel «overlay»[<options>] (<label spec>) at (<coord>) <label text>;` places a label `<label text>` according to the same specifications as if it was passed as a label to an `\informset[color=xg-fg, base line=false, <options>, label=<label spec>]` with first coordinate `(<coord>)`. See the `label` key above for further details.

4.8 Game Trees

The main commands used to draw game trees are the `\informset` and the `\branch` commands. The `\informset` is as described above, except that labels are displayed by default in the `gametree` environment. Specifically, the initial value is `label=135` if the information set is `dashed` or `curved` and it is `label=center` otherwise. The only command left to describe is the `\branch` command.

`\branch «overlay»[<options>] from (<parent>) to[<edge opt>] {<list of children>};` is used to draw an individual branch from the `<parent>` node to the children specified in `<list of children>` on the slides indicated by `<overlay>`. The arguments `<parent>` and `<list of children>` are mandatory in any branch, the argument `<edge opt>` is mandatory for the first branch of a tree, and the arguments `<overlay>` and `<options>` are optional. The `<parent>` has to be a named Ti*k*Z coordinate unless the `root` key is passed in `<options>`, in which case `<parent>` can be any Ti*k*Z coordinate. The `<list of children>` is provided as a semi-colon-separated list, where each `<child>` is described with the syntax `<action>«highlight»[<child opt>](<payoffs>)`. The edge from `<parent>` to `<child>` is labeled by `<action>`. The edge is boldened and highlighted in player i's color (black for player 0) in the slides specified in the overlay specification `<highlight>`. The options `<child opt>` allows fine-tuning of the label and `<payoffs>` will generate a terminal node with label `\ap{<payoffs>}` if `<payoffs>` is a comma-separated list and label `<payoffs>` otherwise. A Ti*k*Z coordinate is placed at the position of child i with name `(<parent>i)` by default or with name `(<parent><action i>)` if the key `index=action` is set. If the `arc=<angle 1>:<angle 2>:<radius>` key is passed to `<edge opt>`, a branch with a continuum of actions is drawn. In this case, the declaration of each child must be preceded by `<angle>:` or `<angle>:<child radius>:`. Doing so places the child at an offset of `<angle>:<child radius>` from the parent node (in polar coordinates), where the default value of `<child radius>` is `<radius>`.

The following key-value pairs can be passed to the `gametree` environment or to `<opt>` in the `\branch` command. They can also be set globally by passing them to `\treesetoptions`.

`alert width` = `<length>` specifies the line width used to highlight edges in the active player's color.

`alt` = `<option>` specifies what is drawn on overlays, on which the branch is not drawn. If `<option>` is empty, nothing is drawn. If `<option>` is `(<list of strings>)` or `()`, a terminal node is placed in the branch's stead with payoffs specified in `<list of strings>` or without terminal payoffs, respectively.

`debug` = `<option>` determines whether the placement of nodes, information sets, and labels are visualized for debugging purposes; see Figure 35. Admissible values for `<option>` are `true`, `detail`, and `false`. The initial value is `false` and the default value is `true`.

`delay contour` = `<bool>` passes on `<bool>` to any nodes, edges, or information sets.

`edge label placement` = `<option>` specifies which algorithm is used to place labels of edges, where `<option>` may take the values `center`, `box`, or `simple`. See the `label placement` key for details.

`edge label sep` = `<length>` specifies the distance of the closest point of the label's bounding box from the outer rim of the edge.

`edge labels` = `<option>` determines the default labels of the tree's edges. If `<option>` is set to `none`, any labels are suppressed. If it is set to `inside` or `outside`, the labels are placed inside or outside of the branch, respectively. The initial value is `outside`.

`font` = `<font spec>` adjusts the font of all labels.

`force math` = `<bool>` determines whether the labels are put into math mode. The initial value is `false` and the default value (if the key is passed without value) is `true`.

`horizontal` = `<bool>` determines whether the tree grows vertically or horizontally, which changes the behavior of the `h` and `v` keys passed to `<edge opt>`; see below for the details. Moreover, the default positions of node labels and terminal payoffs are set to the left and right of the nodes, depending on whether the tree grows left- or rightwards. The initial value is `false` and the default value (if the key is passed without value) is `true`.

`label sep` = `<length>` specifies the distance of the closest point of the labels' bounding boxes from the outer rim of the nodes, edges, and information sets typeset with the `dashed` or `curved` key.

`line width` = `<length>` specifies the line width used to draw edges, nodes, and information sets.

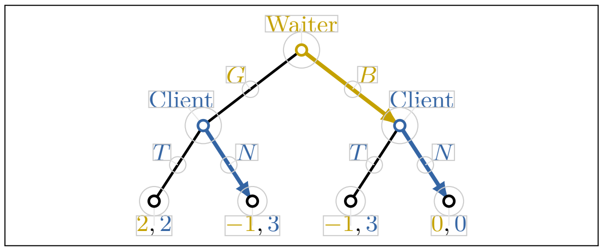
`node label sep` = `<length>` specifies the distance of the closest point of the labels' bounding boxes from the outer rim of non-terminal nodes.

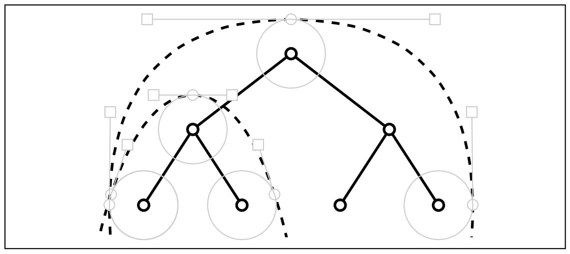
`node radius` = `<length>` specifies the radius of non-terminal nodes.

`offset` = `{<x>,<y>}` offsets the center of the branch from `<parent>` by (`<x>`, `<y>`) in the Ti*k*Z coordinate system.

`payoffs` = `<direction>` specifies the direction, in which terminal payoffs are placed. Accepted values for `<direction>` are the eight cardinal directions `left`, `above left`, `above`, etc. as well as any number, interpreted as angle in degrees.

`player` = `<digit>` sets the active player to `<digit>`. This sets the default color and label to the active player's color and name, respectively, or the active player's number if the key `labels=short` is set. The change of active player is global, hence future calls of `\state`, `\informset`, and `\branch` use the same defaults unless the active player is changed.





```
\begin{gametree}[player names={Waiter, Client}, nodes=hollow, debug=detail]
  \branch[player=1, root=w] from (0,0) to[h=2.6, v=1] {G; B<1->};
  \branch[player=2, label=130] from (w1) to[h=1.3] {T(2,2); N<1->(-1,3)};
  \branch[label=50] from (w2) to {T(-1,3); N<1->(0,0)};
\end{gametree}
\begin{gametree}[labels=none, subgame={debug}]
  \branch[root=w] from (0,0) to[h=2.6, v=1] {G; B};
  \branch from (w1) to[h=1.3] {T(); N()};
  \branch from (w2) to {T(); N()};
  \subgame at (w1);
  \subgame[w=0.99] at (w);
\end{gametree}
```

Figure 35: The `debug` option visualizes that node labels are placed with the `center` algorithm and edge labels are placed with the `box` algorithm by default. Subgames are drawn as (an extension of) a bezier curve around the root of the subgame and its most extreme terminal nodes such that the closest points are at distance `inner sep`. The `debug` option visualizes the control points of the bezier curve and how they are affected by the key `w`.

`terminal = <option>` determines how the terminal payoffs `<payoffs>` are typeset. If `<option>` is set to `p` or `n`, the terminal payoffs are colorized by players. The former value encloses terminal payoffs in parentheses and the latter does not. If `<option>` is set to `e`, labels of terminal payoffs escape the coloring of terminal payoffs. The default values are `n` if `<payoffs>` contains a comma-separated list and `e` if it contains a string without commas.

`terminal label sep = <length>` specifies the distance of the closest point of the labels' bounding boxes from the outer rim of the terminal nodes.

`terminal radius = <length>` specifies the radius of terminal nodes.

`vertical = <bool>` determines whether the tree grows vertically or horizontally, which changes the behavior of the `h` and `v` keys passed to `<edge opt>`; see below for the details. Moreover, the default positions of node labels and terminal payoffs are set above and below the nodes, depending on whether the tree grows up- or downwards. The initial value is `false` and the default value (if the key is passed without value) is `true`.

The following key-value pairs are available only in the `gametree` environment and in `\treesetoptions`.

`edge = <list of keys>` sets `<list of keys>` as default for every edge in the game tree.

`external = <name>` externalizes the resulting image as `<name>.pdf` or the resulting series of images as `<name>-<overlay>.pdf`, depending on the document class. This option requires the `external` package option, which enables Ti*k*Z's externalization library.

`index = <option>` determines how the coordinate at child i's location is named. The coordinate is named `(<parent>i)` if `<option>` is equal to `number` and it is named `(<parent><action`

i>) if `<option>` is equal to `action`. For the latter to work, it is necessary that `<action>` is plain text, that is, it does not contain any Greek letters, any other macros, or any other tokens not allowed in Ti*k*Z coordinate names. The default value is `number`.

`inform = <list of keys>` sets `<list of keys>` as default for every `\informset` in the game tree.

`inform autolabel = <code>` automatically labels information sets in the tree with `<code>`. Use the macro `\pl` in the `<code>` as a placeholder for the player's number and the macro `\enum` as a placeholder for the value of a counter that is increased with each information set. If `<code>` contains the macro `\pl`, player-specific counters are used in `\enum`, whereas a global counter is used if `\pl` does not appear in `<code>`. See Figure 17 for an example.

The `<code>` is fully expanded so that `\pl` and `\enum` are replaced with the correct values. Any non-expandable macros appearing in `<code>` must be protected with a `\noexpand`.

`inform radius = <length>` sets the radius of every player's information sets to `<length>`. Identical radii are typically desired in game trees. If you wish to set different radii for different players, call `\informsetradius` within the `gametree` environment.

`labels = «overlay»<option>` determines the labels of the tree's nodes and the default labels of the tree's information sets. If `<option>` is set to `none`, any labels are suppressed. If it is set to `long`, the default label is the active player's name. If it is set to `short`, the default label is the player's number. The active player is set by the `player` key. The initial value is `long`. Additionally, if an `<overlay>` is provided, labels are displayed only on the specified overlays.

`node autolabel = <code>` automatically labels non-terminal nodes in the tree with `<code>`. Use the macro `\pl` in the `<code>` as a placeholder for the player's number and the macro `\enum` as a placeholder for the value of a counter that is increased with each node. If `<code>` contains the macro `\pl`, player-specific counters are used in `\enum`, whereas a global counter is used if `\pl` does not appear in `<code>`. See Figure 17 for an example.

The `<code>` is fully expanded so that `\pl` and `\enum` are replaced with the correct values. Any non-expandable macros appearing in `<code>` must be protected with a `\noexpand`.

`node label placement = <option>` specifies which algorithm is used to place labels of nodes, where `<option>` may take the values `center`, `box`, or `simple`. See the `label placement` key for details.

`node labels = «overlay»<option>` determines the default labels of the tree's nodes. If `<option>` is set to `none`, any labels are suppressed. If it is set to `long`, the default label is the active player's name. If it is set to `short`, the default label is the player's number. The active player is set by the `player` key. The initial value is `long`. Additionally, if an `<overlay>` is provided, labels are displayed only on the specified overlays.

`nodes = «overlay»<option>` determines the default appearance of nodes in the tree on the slides specified by `<overlay>`. If `<option>` is set to `default` or `full`, the nodes are filled with the active player's color. If it is set to `empty` or `hollow`, the nodes are filled with the package background color `xg-bg`. For options `full` and `hollow`, the node's rim is drawn in the active player's color, for options `default` and `empty`, it is drawn in the package foreground color `xg-fg`. Option `clear` is identical to `empty`, except that labels are suppressed. The initial value is `default`.

`path = <option>` determines how edges are highlighted, where `<option>` is `line` or `arrow`.

`player names` = `<list of strings>` sets the players' names starting from player 1 to the strings in `<list of strings>` and sets the key `players=true`. The player names are set globally, hence they will be available after the `gametree` through the command `\pli`.

`root` = `<option>` specifies the style in which the root of the tree is drawn. Admissible options are `default`, `empty`, `full`, and `hollow` with the identical effect as when they are specified in the `nodes` key. Additionally, `root=clear` resets the special styling of the root node and the same style is used by default for other non-terminal nodes. The initial value is `clear`.

`scale` = `<factor>` scales the game tree by `<factor>`.

`strategy` = `<list of keys>` sets `<list of keys>` as default for every `\strategy` in the game tree.

`subgame` = `<list of keys>` sets `<list of keys>` as default for every `\subgame` in the game tree.

`tikz nodes` = `<bool>` determines whether states are implemented as a full-fledged Ti*k*Z node with path-shortening features or only as a coordinate with a circle-shaped path representation. If the `nonode` package option is specified, the initial value is false, otherwise it is true. The default value (if the key is passed without value) is `true`.

`xscale` = `<factor>` scales the belief space by `<factor>` in the x-direction.

`yscale` = `<factor>` scales the belief space by `<factor>` in the y-direction.

The following key-value pairs are available only as `<options>` in the `\branch` command.

`autolabel` = `<code>` automatically labels all following non-terminal nodes in the tree, starting with the parent of the current `\branch`. Use the macro `\pl` in the `<code>` as a placeholder for the player's number and the macro `\enum` as a placeholder for the value of a counter that is increased with each node. If `<code>` contains the macro `\pl`, player-specific counters are used in `\enum`, whereas a global counter is used if `\pl` does not appear in `<code>`. See Figure 17 for an example.

The `<code>` is fully expanded so that `\pl` and `\enum` are replaced with the correct values. Any non-expandable macros appearing in `<code>` must be protected with a `\noexpand`.

`contour` = `<bool>` determines whether a contour of the package background color `xg-bg` is drawn around the node's label. The initial value is `false` and the default value (if the key is passed without value) is `true`. The contour is drawn using the `contour` package. Setting `contour=delay` is equivalent to setting `contour=true` and `delay contour=true`.

`label` = `<option>` specifies how the label of the `<parent>` node is typeset. The `<option>` can either be `none`, in which case the label is suppressed, it can be a `<float>`, in which case the default player label is set at angle `<float>`, or it can be `<float>:<text>`, which is shorthand for `label=<float>` and `label text=<text>`.

`label text` = `<text>` sets the parent's label to `<label text>`. If parent labels are disabled by default, it is enabled for the current `\branch`.

`node` = `«overlay»<option>` determines the appearance of the `<parent>` node on those slides specified in `<overlay>`. Any defaults specified in the environment key `nodes` are overridden by this key. If `<option>` is set to `default` or `full`, the nodes are filled with the active player's color. If it is set to `empty` or `hollow`, the nodes are filled with the package background color `xg-bg`. For options `full` and `hollow`, the node's rim is drawn in the active player's color, for options `default` and `empty`, it is drawn in the package foreground color `xg-fg`. Option `clear` is identical to `empty`, except that labels are suppressed. The initial value is set by the `nodes` key at the environment level.

`player = <digit>` sets the active player to `<digit>`. This sets the default color and label to the active player's color and name, respectively, or the active player's number if the key `labels=short` is set. The change of active player is global, hence future calls of `\state`, `\informset`, and `\branch` use the same defaults unless the active player is changed.

`root = <string>` specifies that `<parent>` is the root of the tree. The Ti*k*Z coordinates of any node x in the tree will be named `(<string><sequence of actions from root to x>`, where the "actions" in the sequence are either the literal actions if the `index=action` key is set or the index of the actions if the `index=number` key is set.

The above keys `contour`, `delay`, `font`, `label placement`, `label sep`, and `line width` can all be passed to `<edge opt>` in the `\branch` command, in which case they only affect the edges and their labels. Keys `alert width` and `payoffs` can also be passed to `<edge opt>` with the exact same effect as if they are passed to `<options>`. The following key-value pairs can only be passed to `<edge opt>` or they can be set globally by passing them to the `\edgesetoptions` command.

`arc = <angle 1>:<angle 2>:<radius>` is used to draw a branch with a continuum of actions, represented by a sector with radius `<radius>` from `<angle 1>` to `<angle 2>`. If this key is set, the declaration of each child in `<list of children>` has to be preceded by `<angle>:` or `<angle>:<child radius>:`. Doing so places the child at an offset of `<angle>:<child radius>` from the parent node (in polar coordinates), where the default value of `<child radius>` is `<radius>`. By default, the edge label as well as the label of the parent that is drawn next are offset from the sector by a white contour.

`arc color = <color>` sets the color of the sector used to draw continuous-action branches to `<color>`. The initial value is a 10% `xg-fg`.

`color = <color>` sets the color of the edges to `<color>`. The initial value is `xg-fg`.

`e = <bool>` escapes colorizing terminal payoffs if set to `true`. If this option is set, `<payoff>` is typeset directly as the terminal payoff's label. The initial value is `false` and the default value (if the key is passed without value) is `true`.

`etc = <bool>` draws the center half of the edge in a dashed pattern if set to `true`.

`h = <list of num>` is a comma-separated list that determines the horizontal placement of `<list of children>`. If `<list of children>` contains a single child or the tree is oriented horizontally, the horizontal distance from `<parent>` and child i is given by element $\mathrm{mod}(i-1,n)+1$ of `<list of num>`, where n is the length of `<list of num>`. If `<list of children>` contains more than one child and the branch is oriented vertically, children i and $i+1$ are horizontally separated by element $\mathrm{mod}(i-1,n)+1$ of `<list of num>`.

`labels = <option>` determines the default placement of edge labels. If `<option>` is set to `none`, labels are suppressed. If `<option>` is set to `inside` or `outside`, the default position of labels are inside or outside, respectively, of the edges. The initial value is `outside`.

```
\begin{gametree}[node labels=none, inform={curved=15,
↪  label=none}, edge={contour=delay, label sep=1ex}]
  \branch[root=w] from (0,0) to[v=0.8, h=3] {t; n};
  \foreach[count=\i] \s in {+, -} {
    \branch[player=1] from (w\i) to[v=0.9, h=\s1] {B(); N};
    \branch[player=2] from (w\i2) to[h=1] {B(); N()};
  }
  \foreach \x in {, 2} {\informset (w1\x) -- (w2\x);}
  \strategy[prob=1.6]{1/3; 1/3; 1/2}
\end{gametree}
```

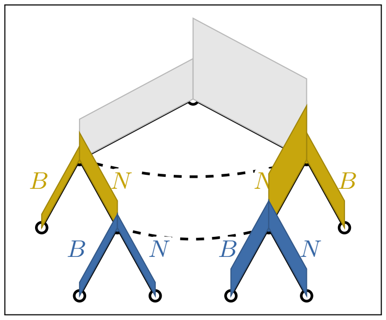


Figure 36: You can stylize individual components of a gametree by setting the default key-value pairs of \informset, \strategy, \subgame, and the to-component of a \branch by passing them to the environment keys inform, strategy, subgame, and edge. This is equivalent to calling \informsetoptions, \strategysetoptions, \subgamesetoptions, and \edgesetoptions inside the gametree environment. In this example, the \strategy is parametrized information set by information set rather than player by player. Note that Player 1's actions have to appear in the same order in (the code of) both \branches for \strategy to match them up correctly, even though B visually appears to the right of the right branch.

n = <bool> colorizes terminal payoffs by players without enclosing them with parentheses if set to true. If this option is set, terminal payoffs are typeset as in \apm. The initial value is false and the default value (if the key is passed without value) is true.

open = <bool> draws the edge in a "trailing-off" pattern to indicate an evergrowing game tree if set to true.

p = <bool> colorizes terminal payoffs by player and encloses them with parentheses if set to true. If this option is set, terminal payoffs are typeset as in \ap. The initial value is true and the default value (if the key is passed without value) is true.

pos = <fraction> sets the default label placement at <fraction> of the way from <parent> to <child>. The <fraction> accepts any formulas that can be parsed by \pgfmathparse.

v = <list of num> is a comma-separated list that determines the vertical placement of <list of children>. If <list of children> contains a single child, the vertical distance from <parent> and the child is given by the first element of <list of num>. If <list of children> contains more than one child and the branch is oriented vertically, the vertical distance from <parent> and child i is given by the negative of element $\mathrm{mod}(i-1,n)+1$ in <list of num>, where n is the length of <list of num>. If <list of children> contains more than one child and the branch is oriented horizontally, children i and $i+1$ are vertically separated by element $\mathrm{mod}(i-1,n)+1$ of <list of num>.

The above keys e, n, p, etc, and open can be passed to <child opt> in the \branch command, in which case they affect only the terminal payoff of exactly that child. Moreover, the following keys can be passed only to <child opt>.

l = <bool> places the edge label of that particular child to the left of the edge *viewed from* <parent> *to* <child>. The default value (if the key is passed without value) is true.

r = <bool> places the edge label of that particular child to the right of the edge *viewed from* <parent> *to* <child>. The default value (if the key is passed without value) is true.

`<float>` places the edge label of that particular child at a fraction `<float>` of the way from `<parent>` to `<child>` if `<float>` is a number in the open interval $(0, 1)$. The initial value is 0.52, which can be changed by providing the `pos` key to `<edge opt>`. If `<float>` is a number outside of $(0, 1)$, terminal payoffs are placed in direction `<float>` of the terminal node. The `<float>` accepts any formulas that can be parsed by `\pgfmathparse`.

Finally, subgames are typeset with the `\subgame` command.

`\subgame «overlay»[<options>] at (<coord>) {<label>};` displays the subgame starting at the coordinate `(<coord>)` on the overlays specified by `<overlay>`. The `<label>` is optional.

The key-value pairs available for `<options>` are the following. Those keys can be set globally with the `\subgamesetoptions` command, for the entire environment by passing them to the `subgame` key of the `gametree` environment, or for each individual `\subgame`.

`color = <color>` sets the color of the `\subgame` and its label to `<color>`. The initial value is the package foreground color `xg-fg`.

`dir = <direction>` indicates in which direction the subgame grows (i.e., the apex lies in the opposite of this `<direction>` from the root of the subgame). Admissible values for `<direction>` are any of the eight cardinal directions `left`, `above left`, `above`, etc. or an angle in degrees. The algorithm will try to determine this direction automatically, but it may need some help for unusual game trees like the signaling game in Figure 19.

`debug = <bool>` indicates whether label placement and the control points of the bezier curve are visualized; see Figure 35. The initial value is `false` and the default value is `true`.

`depth = <dimen>` determines how much beyond the terminal node the subgame extends. The initial value is `2.8ex` so that it includes the labels of terminal nodes in vertically oriented trees.

`font = <font spec>` adjusts the font of the label.

`inner sep = <dimen>` sets the distance of the closest point of the bounding bezier curve from the root and the terminal nodes of the subgame.

`label = <direction>` specifies the direction, in which the label is placed from the apex of the subgame. Admissible values for `<direction>` are any of the eight cardinal directions `left`, `above left`, `above`, etc. or an angle in degrees. The initial value is `135`.

`label sep = <length>` specifies the distance of the closest point of the label's bounding box from the outer rim of the apex of the subgame.

`line = <line spec>` specifies how the line of a subgame is drawn. Allowable values of `<line spec>` are any key-value pair that is understood by the Ti*k*Z draw command. The intended use is for values like `dashed`, `dotted`, etc. The default value is `dashed`.

`line width = <length>` specifies the line width of the subgame.

`w = <option>` determines the shape of the subgame, where `<option>` is a number in $[0, 1)$ or a pair of such numbers, governing each side of the subgame separately. In the coordinate system in the direction of the apex, each control point of the apex is set at `0.8w` of the horizontal distance of the corresponding terminal node. The control point of each terminal node is set at horizontal distance `(1+w)/2` from the apex; see Figure 35.

4.9 Strategies

Strategy profiles can be visualized in two ways with this package. Within a `gametree` environment, the `\strategy` command displays the probability distribution induced over the game tree. The automaton representation of strategy profiles is treated in Section 4.10.

`\strategy` `«overlay»[<options>]{<strategy profile>}` displays the probability distribution induced by `<strategy profile>`. The strategy profile can be parametrized in one of two ways. The first method is to parametrize it by players, then by information sets. In that case, players are separated by ampersands in the input, information sets are separated by semicolons, and the probabilities of actions within the same information set are separated by commas. The second method is to parametrize the strategy profile directly by information sets, regardless of which player they belong to. In either case, information sets and actions have to be specified in the same order in which they were defined using the `\branch` command. The `\strategy` command has to appear in the same `gametree` environment of its corresponding extensive-form game. The `\strategy` command is not compatible with the option `index=action` of the `gametree` environment.

The following options are available for the `\strategy` command.

`positive only = <bool>` indicates whether off-path branches are highlighted using the players colors. If set to `true`, the same effect is generated as if a overlay specification `<overlay>` is passed to `<child opt>` of the `\branch` command.

`prob = <float>` sets the vertical height of one unit of probability to `<float>`.

`root = <name>` specifies that the coordinate `(<name>)` arises with probability 1. The initial value is the root of the tree, but other values can be specified to visualize the conditional distribution in a subgame.

`show prob = <option>` determines whether and on which slides of the frame the probabilities of each action are indicated. Admissible values for `<option>` are `true` and `false`, turning on or off those values for all slides of the overlay, or a comma-separated list of `<strategy options>` indicated below, possibly preceded by an overlay specification. The initial value is `false` and the default value is (if the key is passed without value) `true`.

`show terminal = <option>` determines whether and on which slides of the frame the probabilities of the terminal nodes are indicated. Admissible values for `<option>` are `true` and `false`, turning on or off those values for all slides of the overlay, or a comma-separated list of `<strategy options>` indicated below, possibly preceded by an overlay specification. The initial value is `false` and the default value is (if the key is passed without value) `true`.

The admissible `<strategy options>` are `frac`, which typesets the values in fractional form, or `path`, which suppresses the output off the strategy profile's path. Both options are disabled initially.

4.10 Automata

The main commands used to draw automata are Ti*k*Z's own `\node[atm]` command and the `\edge` command that simplifies drawing arrows in automata. The package also provides the command `\atmstate` to mark up the definition of strategy profiles as in Figure 26.

`\edge` `«overlay»[<options>] <sequence of segments>;` displays state transitions between automaton states. The syntax of each segment in the sequence is of the form `(<from coord>) <direction and labels> (<to coord>)`, where `(<from coord>)` and `(<to coord>)` have to be named coordinates and `<direction and labels>` is either `->`, `<-`, or `<->` with optional labels on either side of the arrows.

The following key-value pairs are available at the environment level or they can be set globally by passing them to `\atmsetoptions`. With the exception of the `radius` key, they can also be provided as `<options>` for individual `\edge`s.

`alert color = <color>` sets the color, with which arrows are highlighted when the `path` key is set to a positive value, to `<color>`.

`ap = <bool>` determines whether edge labels are enclosed with `$\ap$`.

`arrow = <line spec>` specifies how the arrows are drawn. Allowable values of `<line spec>` are any key-value pair that is understood by the Ti*k*Z draw command. The intended use is for arrow shapes like `->`, `-stealth`, etc. The initial value is `-latex`.

`bend left = <option>` specifies whether arrows are curved left. Admissible values are `true` or `false`, which orients all arrows counterclockwise and clockwise, respectively, without changing the curvature, and any `<float>`, which orients all arrows counterclockwise and sets the curvature of non-looping arrows to `<float>`. The initial value is `false` and the default value is `true`.

`bend right = <option>` specifies whether arrows are curved right. Admissible values are `true` or `false`, which orients all arrows clockwise and counterclockwise, respectively, without changing the curvature, and any `<float>`, which orients all arrows clockwise and sets the curvature of non-looping arrows to `<float>`. The initial value is `20` and the default value is `true`.

`curv = <float>` sets the curvature of non-looping arrows to `<float>` without affecting their orientation.

`color = <color>` sets the color of the arrows to `<color>`. The initial value is `xg-fg`.

`debug = <option>` determines whether the placement of the label is visualized for debugging purposes. Admissible values for `<option>` are `true`, `detail`, and `false`. The initial value is `false` and the default value (if the key is passed without value) is `true`.

`font = <font spec>` adjusts the font of all labels in the automaton. The initial value is `\small`.

`force math = <bool>` determines whether the labels are put into math mode. The initial value is `false` and the default value (if the key is passed without value) is `true`.

`label sep = <dimen>` sets the distance at which edge labels are typeset from the arrows to `<dimen>`.

`line width = <dimen>` sets the line width used to draw states and arrows to `<dimen>`.

`loop angle = <angle>` sets the loop radius to a length such that the angle between the beginning and the end of looping arrows is `<angle>`.

`loop radius = <dimen>` sets the radius of any looping arrows to <dimen>.

`path = <float>` specifies the default line width used to draw states.

`pos = <float>` specifies that labels are typeset at a fraction `<float>` along the way of the arrows. The initial value is `0.5`.

`radius = <dimen>` sets the minimum size of the states to `<dimen>`.

Additionally, the `loop` key can be set only for individual `\edges`.

`loop = <direction>` sets this arrow to a looping arrow whose furthest point from the state is in the direction `<direction>`. Admissible values for `<direction>` are the eight cardinal directions `left`, `above left`, `above`, etc. and any angle in degrees.

At the environment level, you can set the various scale keys and enable externalization.

`external = <name>` externalizes the resulting image as `<name>.pdf` or the resulting series of images as `<name>-<overlay>.pdf`, depending on the document class. This option requires the `external` package option, which enables Ti*k*Z's externalization library.

`scale = <factor>` scales the drawn payoffs by `<factor>`.

`xscale = <factor>` scales the drawn payoffs by `<factor>` in the x-direction.

`yscale = <factor>` scales the drawn payoffs by `<factor>` in the y-direction.

Finally, inline automaton states are drawn with the `\atmstate` command.

`\atmstate «overlay»[<options>]{<label>}` displays an automaton state inline with label `<label>`.

Any inline states are affected by the `font` and `line width` key if those are set globally. Additionally, you can fine-tune their appearance with the following keys

`inline radius = <dimen>` sets the minimum size of `\atmstates` to `<dimen>`.

`opacity = <float>` sets the opacity of `\atmstates` to `<float>`.

`xshift = <dimen>` shifts the `\atmstate` by `<dimen>` in the x-direction.

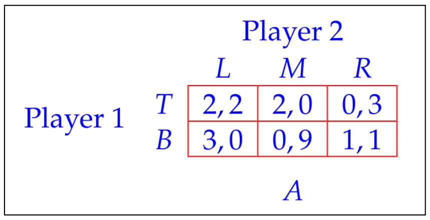
`yshift = <dimen>` shifts the `\atmstate` by `<dimen>` in the y-direction.

5 Ease of Use

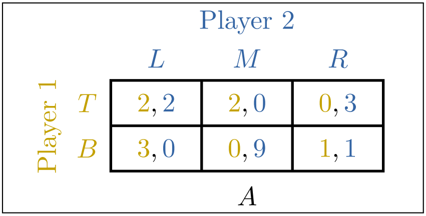
In this section I compare the code and output among the various game theory packages. The `sgamex` and `egameps` are based on `pstricks`, whereas `istgame` and this package are based on Ti*k*Z. As a consequence, `sgamex` and `egameps` compile faster, but trees drawn by `istgame` and this package can easily be marked up with any Ti*k*Z commands.

5.1 Comparison to sgamex

```
% sgamex
\begin{game}{2}{3}[Player~1][Player~2][\(A\)]
  & \(L\) & \(M\) & \(R\)\\
  \(T\) &\(2,2\) &\(2,0\) &\(0,3\)\\
  \(B\) &\(3,0\) &\(0,9\) &\(1,1\)
\end{game}
```

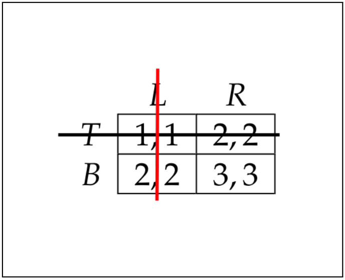

Player 2
L
M
R
Player 1
T
B
2, 2
2, 0
0, 3
3, 0
0, 9
1, 1
A

```
% xgames
\matrixsetoptions{w=1.2, h=0.6, force math}
\begin{matrixgame}[players, top={L, M, R}, left={T, B},
↪  bottom=$A$]
  \payoffmatrix{2, 2 & 2, 0 & 0, 3\\ 3, 0 & 0, 9 & 1, 1}
\end{matrixgame}
```

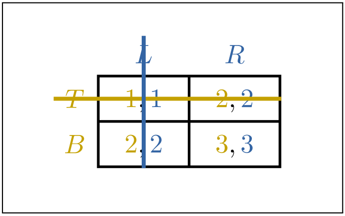

Player 2
L
M
R
Player 1
T
B
2, 2
2, 0
0, 3
3, 0
0, 9
1, 1
A

The above is a comparison of the code and output between the strategic-form game in Figure 4 of the documentation of version 1.0 of the `sgamex` package. For a strategic-form game represented in a single payoff matrix, the code is relatively similar. Whether one prefers the compact input format of a list of optional arguments or a more modular approach using the key-value syntax and/or separate label commands will be a matter of taste. Note, however, that the code doubles with `sgamex` if there are two payoff matrices, whereas only one additional line is needed with this package.

```
% sgamex
\begin{game}{2}{2}
  &\rnode[t]{a12}{\(L\)} &\(R\)\\
  \Rnode[href=20]{a21}{\(T\)} &\(1,1\) &\Rnode{a23}{\(2,2\)}\\
  \(B\) &\rnode[b]{a32}{\(2,2\)} &\(3,3\)
\end{game}
\strike{a21}{a23}
\redStrike{a12}{a32}
```

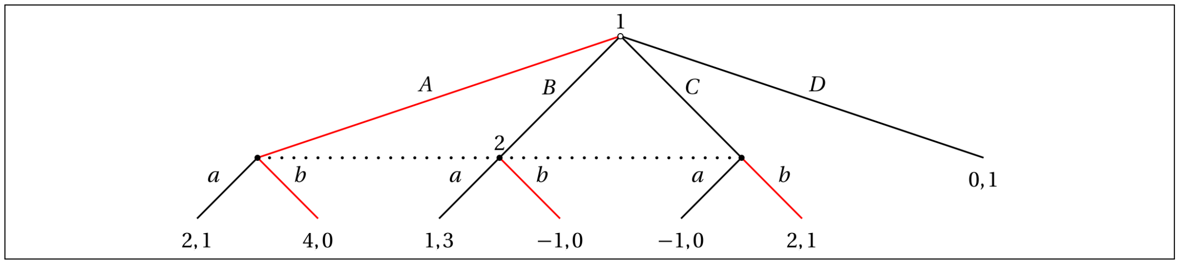

L
R
T
B
1, 1
2, 2
2, 2
3, 3

```
% xgames
\begin{matrixgame}[top={L, R}, left={T, B}]
  \payoffmatrix{1, 1 & 2, 2\\ 2, 2 & 3, 3}
  \strike<2->{h}{1}
  \strike<3->{v}{1}
\end{matrixgame}
```

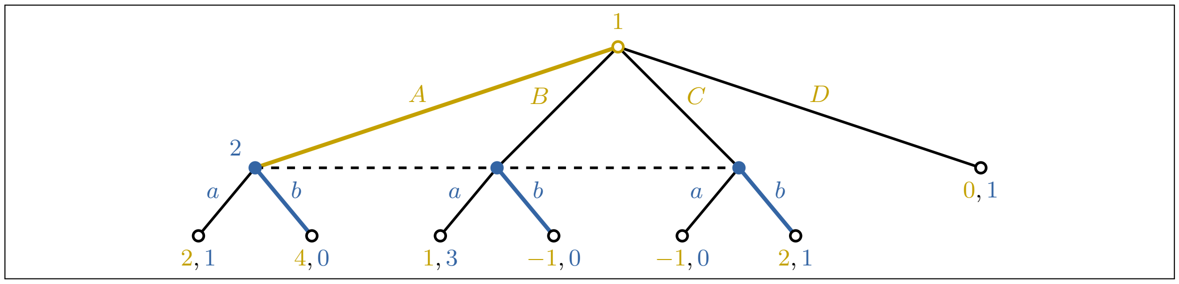

L
R
T
B
1, 1
2, 2
2, 2
3, 3

The above is a comparison of the code and output between the strategic-form game in Figure 10 of the documentation of version 1.0 of the `sgamex` package. When striking through actions, there are some advantages to this package. One need not define the nodes for the strikes manually, the strike command is overlay aware, and if the game is defined by several payoff matrices, the same actions is stricken through in all payoff matrices with a single call.

5.2 Comparison to `egameps`



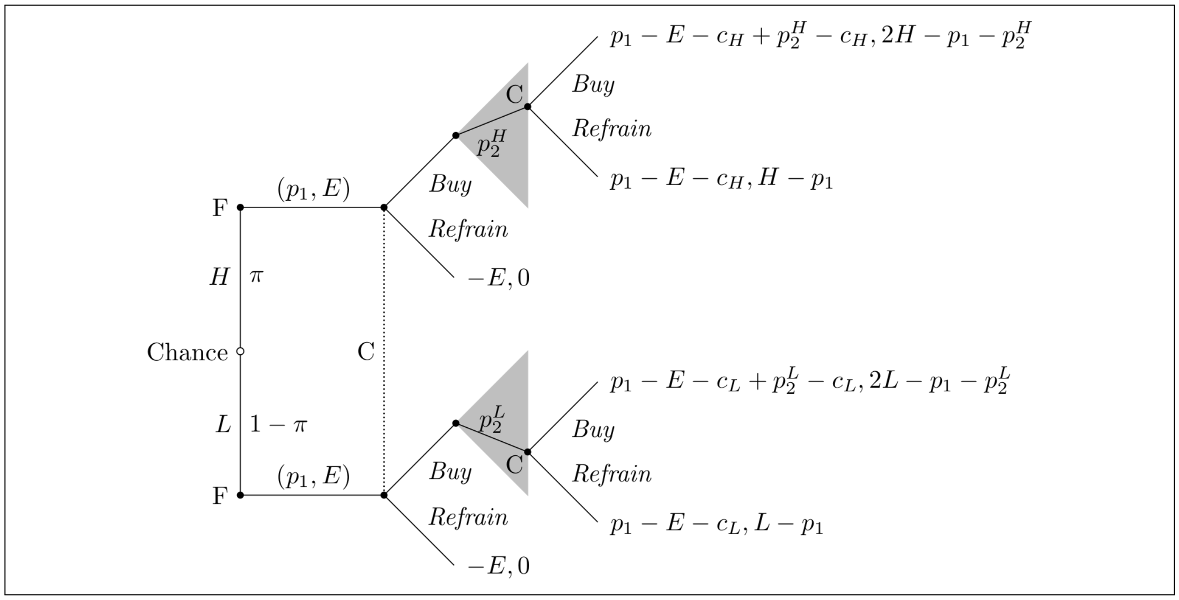
```
% egameps
\begin{egame}(1200,380)
  \putbranch(700,340)(3,1){600}
  \iib[linecolor=red][]{\(1\)}{\(A\)}{\(D\)}[][\(0,1\)]
  \initialtrue
  \putbranch(700,340)(1,1){200}
  \egactionlabelsep=0.5mm
  \iib{}{\(B\)}{\(C\)}
  \egactionlabelsep=1mm
  \putbranch(100,140)(1,1){100}
  \iib[][linecolor=red]{}{\(a\)}{\(b\)}[\(2,1\)][\(4,0\)]
  \putbranch(500,140)(1,1){100}
  \iib[][linecolor=red]{}{\(a\)}{\(b\)}[\(1,3\)][\(-1,0\)]
  \putbranch(900,140)(1,1){100}
  \iib[][linecolor=red]{}{\(a\)}{\(b\)}[\(-1,0\)][\(2,1\)]
  \infoset(100,140){800}{2}
\end{egame}
```



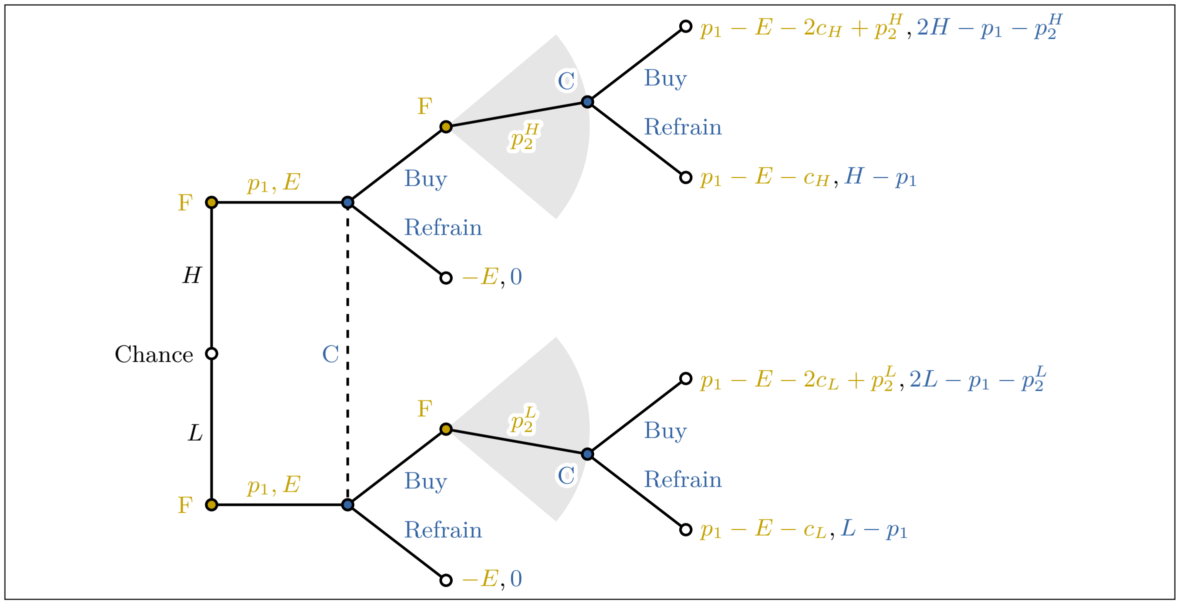
```
% xgames
\begin{gametree}[nodes=full, root=hollow, labels=short, force math]
  \branch[player=1, root=w] from (0,0) to[v=1.8,h=3] {A<1->; B; C; D(0,1)};
  \branch[player=2, label=none] from (w1) to[v=1,h=1.5] {a(2,1); b<1->(4,0)};
  \branch from (w2) to {a(1,3); b<1->(-1,0)};
  \branch from (w3) to {a(-1,0); b<1->(2,1)};
  \informset[dashed] (w1) -- (w2) -- (w3);
\end{gametree}
```

The above is a comparison of the code and output of the extensive-form game in Figure 4 of the documentation of version 1.12 of the `egameps` package. There are numerous advantages to using this package over `egameps`. First and foremost, the coordinates of the individual branches do not have to be computed manually. This makes trees faster to draw and easier to adjust at a later point. Second, writing a tree is more straight-forward since one only ever needs the two rather self-explanatory commands `\branch` and `\informset`. Third, a `\branch` can have any number of children. Fourth, highlighting strategies as well as drawing branches and information sets are all overlay aware. Finally, this package comes with additional features such as adding subgames, displaying the distribution over nodes induced by strategy profiles, and support for Ti*k*Z's externalization library.

5.3 Comparison to istgame



```
% istgame
\begin{istgame}
  \setistgrowdirection'{right}
  \xtdistance{20mm}{20mm}
  \istroot(0)[chance node]<180>{Chance}
  \istB<grow=north>{H}[l]{\pi}[r] \istB<grow=south>{L}[l]{1-\pi}[r] \endist
  \istroot(H0)(0-1)<180>{F}
  \istb{(p_1,E)}[a] \endist
  \istroot(L0)(0-2)<180>{F}
  \istb{(p_1,E)}[a] \endist
  \setistmathTF*001
  \cntmdistance*{10mm}{20mm}{8mm}
  \istroot(H)(H0-1)
  \istb{Buy}[br] \istb{Refrain}[ar]{-E,0} \endist
  \istrootcntm(H1)(H-1)
  \istb{$p_2^H$}[b] \istbm \endist
  \istroot(CH)(H1-1)<[label distance=-4pt]135>{C}
  \istb{Buy}[br]{p_1-E-c_H+p_2^H-c_H, 2H-p_1-p_2^H}
  \istb{Refrain}[ar]{p_1-E-c_H, H-p_1}
  \endist
  \istroot(L)(L0-1)
  \istb{Buy}[br] \istb{Refrain}[ar]{-E,0} \endist
  \istrootcntm(L1)(L-1)
  \istbm \istb{$p_2^L$}[a] \endist
  \istroot(CL)(L1-2)<[label distance=-4pt]-135>{C}
  \istb{Buy}[br]{p_1-E-c_L+p_2^L-c_L, 2L-p_1-p_2^L}
  \istb{Refrain}[ar]{p_1-E-c_L, L-p_1}
  \endist
  \xtInfoset(H)(L){C}[left]
\end{istgame}
```



```
% xgames
\setplayernames[0]{Chance, F, C}
\begin{gametree}[horizontal, edge labels=inside]
  \branch[root=w] from (0,0) to[h=0, v=4.2, labels=outside] {$H$; $L$};
  \foreach[count=\i] \x in {H, L} {
    \branch[player=1] from (w\i) to[h=1.8, v=0, labels=outside] {$p_1, E$[0.46]};
    \branch[player=2, label=none] from (w\i1) to[h=1.3, v=2.2] {Buy; Refrain(-E, 0)};
    \branch[player=1, label=135] from (w\i11) to[arc=40:-40:1.9] {30-20*\i:$p_2^\x$[0.45]};
    \branch[player=2, label=405-270*\i] from (w\i111) to[h=1.3, v=2]
    ↪  {Buy(p_1-E-2c_{\x}+p_2^{\x}, 2\x-p_1-p_2^{\x}); Refrain(p_1-E-c_{\x}, \x-p_1)};
  }
  \informset[player=2, dashed, label=0.5] (w11) -- (w21);
\end{gametree}
```

The above is a comparison of the code and output of the extensive-form game in Section 16.10 of the documentation of version 2.0 of the `istgame` package. The code is somewhat shorter with this package, even after discounting the for loop in the second piece of code. An advantage of this package is again that one only needs the two intuitive commands `\branch` and `\informset` to draw the tree. Moreover, all commands are overlay aware and coloring as well as labeling is automatic and that those commands are overlay aware. Moreover, the key-value syntax may be easier to use than commands with a long sequence of parameters. A distinct advantage of the `istgame` package is that it compiles quicker. Drawing the above tree takes 0.151 seconds with `istgame` and 0.246 seconds with this package.[12]

[12]The test was performed over an average of 100 runs on a i9-10900K processor with the code by Phelype Oleinik at `https://tex.stackexchange.com/a/505840/203616`.

6 Frequently Asked Questions

I get a whole bunch of `\GenericErrors` or other confusing errors.

There are three likely causes for this:

- You are using a non-expandable macro in an `autolabel` or in any label within an `\xforeach` loop. In this case, the solution is to precede all non-expandable macros by a `\noexpand`.
- The actions specified in a `gametree` involve characters that are not allowed in the name of a Ti*k*Z node while the `index=action` is active. The easiest solution is to set `index=number` instead.
- There is a bug in the package's code. Even though I may not always respond immediately, I appreciate emails with bug reports because I want this package to be bug free eventually.