Graphical User Interface for Parity Games: A Tool for Algorithm Visualization and Interaction

K. Mitka, Y. Mashal, H. Stokman, M. Vanoušek, H. T. Bui

April 19, 2024

Contents

1 Introduction

Parity games are infinite-duration games, defined by directed graphs with a weight (priority) and a player (Even or Odd) assigned to every node. They are notably used in the verification of reactive systems, model checking, and the synthesis of controllers, making their study interesting and necessary. Despite the existence of numerous algorithms aimed at finding winning strategies for parity games, a common bottleneck is their exponential complexity in worst-case scenarios. Therefore an important part of parity game research is to find counter-examples and worst-case scenarios for these algorithms. Currently, researchers do this by drawing the graph by hand on a whiteboard and manually going through the steps of an algorithm. We aim to improve this process with this tool.

Commissioned by Tom van Dijk, this project seeks to bridge the gap between theoretical algorithm analysis and practical application through the development of a Graphical User Interface (GUI) tool. This tool aims to facilitate the creation, visualization, and manipulation of parity game diagrams, offering an interactive platform for researchers and educators alike. The tool also allows the user to easily visualise the steps an algorithm goes through to solve the parity game. By streamlining the process of diagram creation and enhancing the visualization of algorithmic steps in parity game analysis, the tool endeavours to boost the efficiency of research in this domain.

2 Requirement Specification

For any project, a well-defined set of requirements is crucial to ensure clarity and focus during the development process. This section lists all the functional and quality requirements, which are prioritised using the MoSCoW method.

The priorities were assigned by keeping in mind the different target users of the app, and the limited time, since we have had 10 weeks to design and develop the full product. We will talk about the target users more in detail after listing the requirements.

2.1 Functional Requirements

Functional requirements specify the fundamental actions that the system must perform. For this project, they include the following:

2.1.1 Streamline the process of creating parity games and their diagram representation

- The program **must** save a parity game diagram to a file. The format of this file needs to be designed by us.
- The program **must** load a parity game diagram from a saved file.
- The program **must** export a diagram to a parity game file format supported by Oink.
- The program **must** import a parity game file supported by Oink and create a diagram.
- The program **must** display a parity game diagram.

- The program **must** be able to perform basic changes to the parity game diagram.

  These include:

  - Add or remove a node.
  - Change to which player a node belongs.
  - Increment, decrement or set the priority of a node.
  - Add or remove an edge.

- The program **should** allow selecting multiple nodes or multiple edges at a time and performing operations on all of them.

- The program **should** be able to perform actions to change the parity game diagram appearance.

  These include:

  - Change the position of a node relative to the rest of the nodes.
  - Change the shape of an edge.

- The program **should** allow automatically rearranging the relative positions of a group of nodes (potentially all the nodes), to create a visually pleasing diagram.

- The program **should** allow the user to undo and redo any change made in the parity game diagram.

- The program **could** allow the user to remove a node while connecting each parent to all its children.

- The program **could** have the ability to clone a group of nodes, such that:

  - Each original node receives a clone with identical priority.
  - The clones are interconnected as the originals.
  - For each parent of an original node, an edge is added from it and the original node's clone.
  - For each child of an original node, an edge is added from the cloned node to the child.

- The program **could** allow grouping nodes to highlight the parity game's hierarchy.

- The program **could** allow subdividing an edge.

- The program **could** allow dragging an edge uw onto a node v, removing uw and adding uv and vw.

2.1.2 Visualize steps of parity game algorithms

- We **must** define a trace file format, which allows visualization of algorithm execution.

- The trace file format **must** be well documented, to allow researchers to implement programs that generate it while solving a parity game.

- The file format **should** allow highlighting sets of nodes and sets of edges.

- The file format **should** allow setting a label to a node.
- We **should** implement at least one parity game algorithm with trace as output.
- We **should** allow the user to submit their own parity game algorithm, which generates a trace in JavaScript. This will facilitate quick iteration of the parity game design based on the algorithm's trace.
- The GUI **should** allow the user to select which stets of nodes/edges to highlight while stepping through the trace.
- We **could** implement parity algorithms Zielonka, small progress measures, and priority promotion with a press of a button in the GUI.
- The colour scheme for highlighting **won't** be customizable.

2.2 Quality Requirements

Quality requirements define the system's quality attributes and operational characteristics.

- **Usability:**
  - The program **must** have a manual, which allows a student or a researcher to start using the program without help within 5 minutes.
  - The program **must** have convenient keyboard shortcuts for the most commonly performed tasks.
- **Performance:**
  - The program **should** parity games with up to 150 nodes smoothly on average hardware, ensuring responsiveness and minimizing lag during interactions.
- **Reliability:**
  - The program **must** be robust, with minimal bugs and crashes, ensuring a consistent user experience.
  - The program **should** include an autosave feature to prevent data loss.
- **Accessibility:**
  - The program **should** be available on a wide variety of devices and operating systems.
- **Security:**
  - The program **must** ensure the security of user data and prevent code injection when importing a parity game file.
  - The program **must** warn users about the risks of running custom parity game solvers on their computers and the necessity to inspect their source code before doing so.

2.3 User Types

Our app will be primarily used by two different types of users. The first type of user is a researcher, who wants to easily display and edit parity games, and view and analyse the behaviour of various parity game solvers. The second type of user is a student learning about parity games, who wants to gain insight into how the parity game solvers work.

These targets do differ in a few ways. For example, researchers mostly want a powerful tool with many features, and would not mind it too much if the tool takes some time to learn. Students might not care as much about the capabilities, but mainly want the tool to be easy to use, so they can play around with parity games without needing to learn much about the tool itself. We mainly focused on the researchers, as this was said to be the primary target user by the client, but we also kept the students in mind. For example, having the in-app manual is convenient but not that important for researchers, who can just look at the README on GitHub when they start using the app. It is however an important requirement for students who don't use the app as much as a parity game researcher would.

3 Global Design

In this section, we will explain some of the high-level decisions we made like the platform and technologies we decided to use and what our vision is for how the application is going to be used. This is mainly related to what things we deem as important, which will guide our decision-making on how features should be implemented to best fit these values.

3.1 Web app vs Native app

In our case, portability was an important requirement. Researchers and students tend to use a wide variety of devices. Our team alone runs Windows, Linux and macOS. Furthermore, we believe that not having to install the app will make it more likely for students of a parity game course to use it. For those reasons, we decided to build the app in the browser. Click here to try it out!

Even while building our application for the browser, we still had some OS issues. We tested the app on Linux, but even in the "standardised" environment of the web browser, Tom's Linux Mint distribution still behaved differently. We would likely have a lot more of these issues for a native app.

3.2 Static website

We decided to keep our application a static website instead of having a separate server side with a database. We feel like this is a worthwhile tradeoff with minimal downside. The main benefit of a non-static webpage would be the ability to have user accounts, where users could store parity games for easy access and as a backup. Instead, we rely on the user's storage system, supplemented with the browser's local storage, to save their work and we keep the editor more lightweight, just focusing on editing and visualisation. We did this for the following upsides of a static page:

- Easy of hosting/development: Having a static web page makes our hosting very simple since we just need to serve once upon loading. This is not only good for the reliability of our application

for the user since there are fewer points of failure, but also makes development a lot easier. You can simply run the application locally without needing to run a database/server.

- Easy to use: Users are not required to make an account and can immediately use the site when visiting.
- Security: With a static web page, there are fewer possible security vulnerabilities to worry about. We still sanitise importing of files, but any security breach is more limited in scope as there is no back-end that could get compromised.

3.3 Technology Stack

Before the work on any software application can begin, developers must choose the technology they will use to build it.

3.3.1 TypeScript

TypeScript has been selected as the primary language due to its strong typing system, which enhances code quality and understandability. Its strong typing system reduces the likelihood of runtime errors, making the development process more efficient and the code more reliable. For some members of our team, this will also be a valuable introduction to this technology, which is prevalent in the industry. TypeScript's compatibility with JavaScript also ensures wide support and the ability to leverage the vast ecosystem of JavaScript libraries and tools.

One of these JavaScript libraries that we used is Cytoscape.js. It is a library for visualizing graphs, which already allows for rendering, selecting and moving nodes/edges out of the box. It also has many extensions that work with it. For example, we used the cytoscape-edge-editing library which allows users to create bend or control points on edges.
However, some of these libraries did not support types, which made it a hassle to use them with TypeScript. We did find out how to resolve this issue, so we do not regret choosing TypeScript over JavaScript, but it was a hurdle that we did not anticipate when making the choice.

3.3.2 Webpack

Since we already need to compile TypeScript, we use Webpack to bundle the files. Bundlers, such as Webpack, combine the individual JavaScript files into one, which leads to improved load times for our app.

3.3.3 Github Pages

GitHub Pages has been chosen for hosting due to its simplicity, cost-effectiveness (free for public repositories), and seamless integration with our development workflow. It is an ideal solution for our web app's static pages, allowing for quick deployments and updates directly from our GitHub repository. GitHub Pages' support for custom domains and its robust infrastructure ensure our web app is reliable and accessible.

3.3.4 GitHub CD

Finally, we set up a Continuous Delivery (CD) pipeline. We chose GitHub Actions for this because it's integrated right into GitHub, making our development process smoother and faster. It automates our build, test, and deployment workflows, catching bugs early and speeding up releases. Plus, it's a great way for the team to get hands-on with CI/CD practices, boosting our skills in modern software development. It's flexible, supports a wide range of tools, and keeps everything in one place, simplifying our project management. This also ensures that every time we merge to main, the changes will be deployed to GitHub Pages within a few minutes. We hope this will facilitate getting frequent and up-to-date feedback from the client.

3.4 Page Design

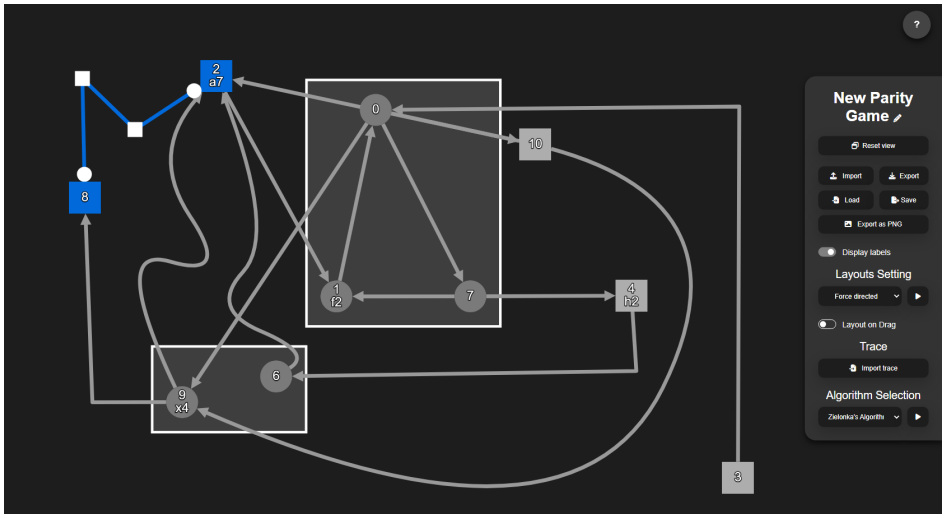

Figure 1: Graph editing example

3.4.1 Single Page

We decided to put our entire application on a single page. This removes the need to navigate to different pages to find particular features. The webpage has two sections: the parity game itself and a sidebar with buttons for some of the features.

3.4.2 Minimal UI

Another goal we had was to make the GUI look very minimal, while still packing a lot of features into the GUI. One way we did this is through keybinds, but also through UI elements that only appear when needed. For example, the GUI for the algorithm trace only appears after importing a trace and doesn't take up space before that. As shown in Figure 1, there is no UI space dedicated to graph editing. We do not have any buttons for features like adding nodes or edges, changing the shape of

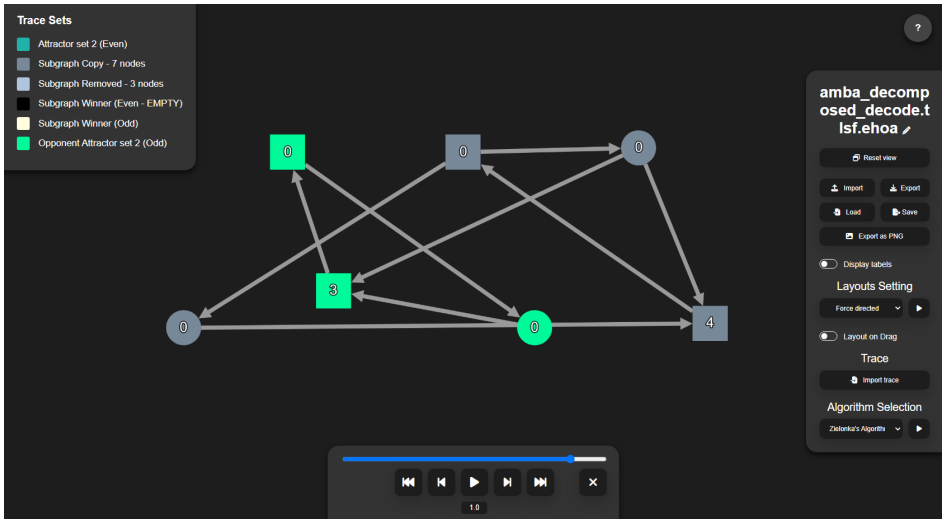

Figure 2: Algorithm trace UI

edges, or changing the priority of nodes. Instead, we use keybinds for these features. This way, most of the space on the page is dedicated to displaying the graph, apart from one navigation bar with some features. It also allows users to edit the graph much faster once they get acquainted with the keybinds. Since the main purpose of the application is graph editing, the speed and comfort with which the user edits is very important.

It does require the user to read a manual on how to use the program. To help the user to learn the application quickly, the manual is accessible within the app, and the keybinds are made as intuitive as possible. For some actions, we use keybinds that are used for the same purpose in a lot of other applications. For other keybinds, we tried making them mnemonics. For example, 'l' is used to change the label of a node. This way the user still needs to read the manual to use the application but will not have to put much effort into remembering the keybinds. We will go more into detail about this in the next section.

4 Detailed Design

4.1 Simple parity game changes

For basic changes to the parity game diagram, we made keybinds for every action. We also created a manual that shows all of these keybinds to the user. When choosing what key binds an action should have, we decided to use keys that are intuitive to the user. However, after feedback from our client, we realized that for an experienced user, it would be more ergonomic and faster to have keybinds within reach of your left hand. This is why we have multiple keybinds for some features, as we wanted to incorporate this while also keeping the mnemonics for the students.

A node is created with either 'e' (for even) or 'o' (for odd), depending on who should own the

node. An odd node can also be created with 'w', as 'o' is too far away but is a good mnemonic. The owner of selected nodes can then be changed by pressing 'q'. The priority of selected nodes can be incremented with the '+' and '2' keys, decremented with the '-' and '1' keys, and set to any number with the 'p' and 'x' keys. The label of selected nodes can be changed with the 'l' and 'c' keys. We believe these keybinds make enough sense for users to easily remember them after reading the manual.

Multiple nodes and/or edges can be selected using shift: either by dragging to select all in an area or by clicking on the individual elements. Selected elements can be deleted with Delete or Backspace, copied with Ctrl+C or Cmd+C, and pasted with Ctrl+V or Cmd+V. Any change made on the diagram can be undone with Ctrl+Z or Cmd+Z, and redone with Ctrl+Y or Cmd+Y. These Ctrl/Cmd, Shift and Delete keybinds are standard behaviour for a lot of applications, and shouldn't come as a surprise to most users. As the Delete and Backspace keys are too far away from the rest of the keys, elements can also be deleted by pressing 'd'.

All in all, we made keybinds for these requirements because this allows the user to use these features much faster once they learn the keybinds. It does not require much time to learn all of these, and by not having any buttons for these features it keeps the screen clean, while also saving some development time in adding UI for these actions.

There are also some buttons shown when right-clicking an edge. There are options to create bend or control points. If they already exist on that edge, there is an option to remove all of them. This allows the user to freely shape their edge the way they want it. When right-clicking a node, an option is shown to set the priority to a certain number. This sets the priority of all selected nodes.

We do display buttons on the screen for some more advanced changes on the diagram. Most of these we display on the right side of the screen. These are mainly changes that impact the entire diagram, and not just a single node or edge. One of these is a button that toggles the display of labels in the parity game.

4.2 Automatic Layout

On the sidebar, there is an option to run an automatic layout algorithm to determine the positions of the nodes. There are a few standard options available. The options are force-directed, grid, breadth-first and random.

The grid layout places the nodes in a grid, with the same distance between them. They are placed based on the IDs of the nodes, so the layout does not account for where the edges are.

The breadth-first layout places the nodes in different levels, as generated by running a breadth-first search on the graph.

The force-directed layout places the nodes based on a simulation of forces. We use the cola.js extension for Cytoscape.js to run this layout, as it offers a lot of customisation options and is well suited

for successive runs of the layout.

The random layout places all nodes randomly. It is meant as more of a fun way to play around with the layout, as there is no guarantee it creates any structure.

There is a toggle button shown for rerunning the layout whenever the user is dragging a node. This button is only shown if the force-directed layout is selected, since the grid and breadth-first layout stay the same and don't allow for dragging nodes, and the random layout will just randomly assign the positions of the nodes on every mouse movement when dragging. The cola.js force-directed layout works well with dragging nodes. It keeps simulating while dragging, which allows the user to have some control over the force-directed layout.

Nodes can be grouped together using 'g'. This fixes their position and causes the automatic layout to ignore this group and only change the position of the other nodes. This way the user can try the automatic layout on some nodes while keeping the ones the user is already happy with in the same place. It can happen that running the automatic layout causes a node to overlap with a node in such a group, but the user can easily move the entire group of nodes out of the way to resolve the issue.

4.3 Saving & Sharing

Since parity game researchers have already made a lot of parity games with Oink, it is important to have good import functionality so they don't have to recreate the diagram manually. There is a button for importing a file into the app. This file has to be a .pg file with standard format (as defined by Oink) to be able to be imported. Since this file definition only contains the mathematical definition of the graph and does not contain the position of the nodes, the current layout setting is automatically run to display the parity game. There is also an option to export the graph back to this same .pg format.

The application also allows saving parity games. This saves the full JSON that is used to render the graph, which means that the layout is also saved in this file format. There is also an option to load a parity game that has this format.

There is also an option to export the diagram to a PNG. This allows the user to easily save the parity game as a picture without having to screenshot it themselves.

The application also autosaves the graph state in localstorage. This way the progress of the user is not lost when accidentally closing the tab or refreshing the page.

4.4 The Algorithm Trace

The algorithm trace feature allows the user to visualize the step-by-step execution of a parity game solver. There are two main reasons one would use this feature: First, a student of a course about parity games can use it to better understand the way a parity game solver works. Second, a researcher can use this feature when analysing the behaviour of algorithms, for example when trying to create a parity game which has exponential time complexity for a certain solver.

The steps of any parity game solver can be visualised. For this, an implementation of the algorithm that produces a trace is needed. This trace is a list of steps, where each step highlights some groups of nodes or edges. For example, the last step in an algorithm could highlight the nodes that belong to the winning region of the odd and even player respectively (two node sets and zero edge sets). Also, each step can add labels to individual nodes.

The Zielonka algorithm can be run out of the box just by pressing a button. We offer two ways to use the app with custom algorithms: either implement the algorithm natively in TypeScript or use a language of your choice to generate the trace file and import it. Read the developer documentation for more details.

4.4.1 Trace File Format

The trace file format is a custom JSON format designed to encompass the essential states and transitions involved in the computation of a parity game solution. For each step in the trace, a list of node sets and edge sets are given by IDs, to highlight different groups of nodes or edges in every step.

The trace also stores a description of the parity game it was created for, to make sure the parity game shown in the application is the same as the game that the algorithm ran on. If these games differ, we show a popup that allows the user to overwrite their current parity game with the one of the trace file.

4.4.2 Visualisation Mechanics

The GUI component responsible for trace visualisation reads the trace file and sequentially displays each step. Users can navigate through steps using forward and backward controls, allowing them to analyze the algorithm's progress at their own pace. The trace steps can also be played automatically with an adjustable speed setting. When on a certain step, any trace sets can be disabled by clicking on the square icon of the set.

5 Testing

5.1 Start Early

Setting up a testing environment in a project is a task prone to procrastination. It is not as exciting as adding features to the app. The longer the addition of the testing framework is delayed, the harder it becomes to add enough tests. As a result, if an app isn't tested from the very beginning, the tests are often never added.

For this reason, before we developed the first feature, we set up a testing environment for the project using the Jest framework. Of course, all tests should be run before any code is merged into the main. People are fallible and tend to forget these things, so we created a Github Action, which will run all tests on every open merge request and require these checks to pass before any changes can be merged to the main branch.

5.2 Unit Tests

Having the Jest framework available in our project removed friction in writing unit tests for the more complicated functions. While we did not have many complex functions, we tested those which seemed to warrant this approach, such as importing and exporting .pg files, serializing and deserializing the Trace, etc.

These unit tests helped with the smooth deployment of features. Bugs in the parsing were discovered before any attempt was made to integrate them into features. On one occasion, the refactor broke the parsing, which was caught by the test when a pull request for the change was opened. But even this testing has its limitations. A portability bug in the parsing function was only detected after the pull request was merged to the main branch because it only occurred on Windows – the developer and the CI CD pipeline both ran the test on Linux.

5.3 Feature & Integration Tests

The reason we picked Jest as our testing framework is because it allows easy "visual" testing of websites. Its snapshot feature allows the framework to "remember" what a certain state of the app should look like (both content and styling) and fail the test if it doesn't find what it expects. The developer can then look at the before and after appearance and decide if this is an issue or desired change.

Early in the development, we introduced these visual tests. But we quickly discovered that this early in the development process, they tend to get in the way of progress. Almost any change was meant to be a change of the UI. Any hope of catching unintended changes quickly vanished as the developers acquired the habit of quickly marking all changes as intended. As such, this visual testing was seen as an impediment to productivity and did not fulfil the goal of increased reliability. We therefore removed these tests and relied on manual feature and integration tests. As the app matures and becomes stable in appearance, however, it quickly becomes more sensible to bring these tests back.

6 Future Work

Due to time constraints, we did not implement all the requirements. However, we are happy with what we were able to do since we implemented all the **must** and **should** requirements.
As for the **could** requirements, we did not implement the following:

- The program **could** allow the user to remove a node while connecting each parent to all its children.
- The program **could** have the ability to clone a group of nodes, such that:
  - Each original node receives a clone with identical priority.
  - The clones are interconnected as the originals.
  - For each parent of an original node, an edge is added from it and the original node's clone.
  - For each child of an original node, an edge is added from the cloned node to the child.
- The program **could** allow subdividing an edge.

- The program **could** allow dragging an edge uw onto a node v, removing uw and adding uv and vw.

Other than the requirements we did not implement, there are also some features of the system that could be improved in further development. For the automatic layout, an option for a planar embedding could be added. It is possible to determine whether a graph is planar and to construct a planar embedding for it if it is indeed planar. This would create a diagram that is easy to interpret for the user. In general, the options for the automatic layout could be improved, as they currently don't provide much clarity in big graphs.
Another potential improvement is to allow the user to save to any location, instead of always saving the file in the Downloads folder when saving or exporting the parity game.

7 Source Code and Manual

The GitHub repository can be found here. The repository's README contains documentation for developers on how to build the project and instructions for adding algorithms for trace generation.

The application manual can be found in the application by clicking on the '?' in the top right corner or pressing the '?' key.

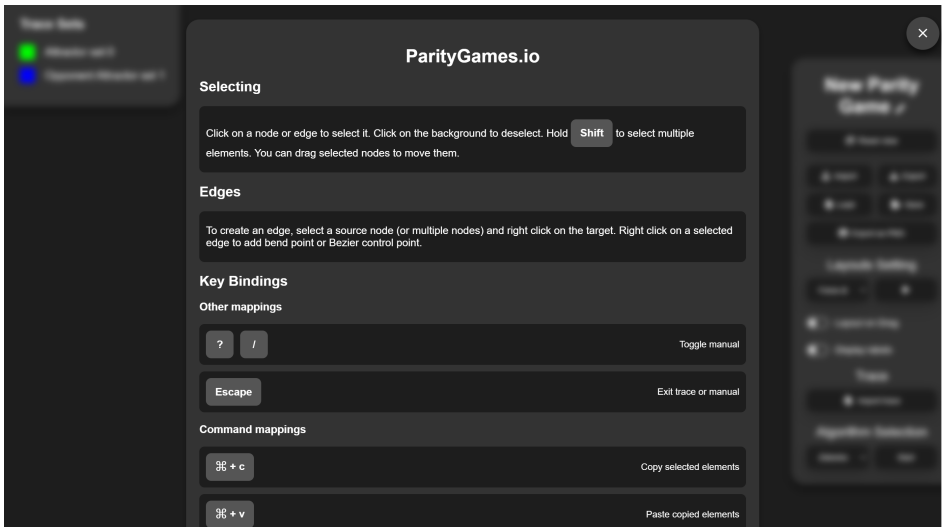

Figure 3: Application Manual

8 Individual Contributions

Below is a table with the most important contributions of each team member.

Name	Contribution
Miki	• Setup Project (Github repository with CI for testing, Build system with Webpack) • Create tests • Parsing .pg file • Create trace UI • Keyboard Shortcuts • Documentation (README and Manual) • Clean up code structure
Krystof	• Add support for Algorithms • Implement Zielonka's algorithm with trace • Improve Copy/Paste • Import JavaScript libraries without type support • Improve GUI look • Make application PWA and downloadable • Clean up code structure
Han	• Implement Automatic Layout options • Implement Layout on Drag • Implement Grouping of nodes • Import/export oink file and with visuals • Navigation Bar

Name	Contribution

Hessel	• Setup Project with Cytoscape.js
	• Basic Graph editing
	  – Creating even/odd nodes
	  – Creating edges
	  – Multiselect nodes/edges
	  – Deleting nodes/edges
	• Copy/Paste
	• Change Priority and Label of nodes
	• improve GUI look
	• Write Design Report

Yazan	• Autosave • Importing and playing Trace • Create trace UI • Editing parity game name • Resetting graph view

Niels	• Basic Graph editing
	  – Disallow multiple edges
	  – fix Copy/Paste
	  – Node labels
	• Bend and Control points for edges
	• Make many actions undo/redoable
	• Bugfixing and manual testing
	• Write Design Report

9 Conclusion

In conclusion, the development of a parity games GUI has been successful, and we think it will be greatly beneficial to parity game researchers and students alike. There remains room for future improvements to the system, but we are happy with the features we managed to implement given the time constraints.