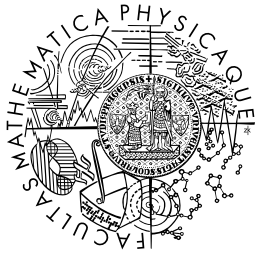


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Tomáš Čížek

Implementation of the Sprouts game

Department of Applied Mathematics

Supervisor of the bachelor thesis: RNDr. Martin Balko, Ph.D.
Study programme: Programming and Software Systems
Study branch: Computer Science

Prague 2021

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to express my sincere gratitude to RNDr. Martin Balko, Ph.D for providing continuous and highly-valued assistance together with his endless enthusiasm. I would also like to thank to doc. RNDr. Pavel Valtr, Dr. for his support from the grant no. 21/32817S of the Czech Science Foundation (GAČR). Last but not least, I thank my family for supporting me throughout the work on this thesis.

Title: Implementation of the Sprouts game

Author: Tomáš Čížek

Department: Department of Applied Mathematics

Supervisor: RNDr. Martin Balko, Ph.D., Department of Applied Mathematics

Abstract: Sprouts is a two-player pencil-and-paper game invented by John Conway and Michael Paterson in 1967. In the game, the players take turns in joining dots by curves according to simple rules, until one player cannot make a move. The Game of Sprouts is very popular and simple-looking, so it may come as a surprise that there are essentially no AI Sprouts players available. This lack of computer opponents is caused by the fact that the game hides a surprisingly high combinatorial complexity and implementing it involves fascinating programming challenges.

We overcome all the implementation barriers and create the first user-friendly Sprouts application with a strong artificial intelligence after more than 50 years of the existence of the game. In particular, we combine results from the theory of nimbers with new methods based on Delaunay triangulations and crossing-preserving force-directed algorithms to develop an AI Sprouts player which plays a perfect game on up to 11 spots.

Keywords: Sprouts, Combinatorial game, Planar graphs, Artificial intelligence

Contents

1 Introduction **3**
- 1.1 History of Sprouts . . . 4
- 1.2 Theory behind Sprouts . . . 5
  - 1.2.1 Terminology . . . 5
  - 1.2.2 Basic properties . . . 6
  - 1.2.3 Game strategy . . . 8
  - 1.2.4 Sprouts variations . . . 9
- 1.3 Existing implementations . . . 10
- 1.4 Our goals . . . 14

2 Analysis **16**
- 2.1 Graphical representation . . . 16
  - 2.1.1 Components of the graphical representation . . . 17
  - 2.1.2 Drawing a move . . . 20
  - 2.1.3 Inserting a move . . . 21
- 2.2 Redrawing algorithm . . . 24
  - 2.2.1 Types of forces . . . 25
  - 2.2.2 Moving a vertex . . . 27
  - 2.2.3 Improvements . . . 28
  - 2.2.4 Summary . . . 31
- 2.3 Computer opponent . . . 32
  - 2.3.1 String representation . . . 34
  - 2.3.2 Simplifying strings . . . 36
  - 2.3.3 Generating children of a position . . . 40
  - 2.3.4 Computation of outcomes . . . 41
  - 2.3.5 Parallel exploration . . . 46
  - 2.3.6 Training of the computer opponent . . . 48
- 2.4 Synchronization of the representations . . . 51
  - 2.4.1 Move graphical representation . . . 52
  - 2.4.2 Drawing a computer move . . . 53
  - 2.4.3 Parallel representations . . . 61

3 Implementation **63**
- 3.1 Used technologies . . . 63
- 3.2 Solution structure . . . 63
- 3.3 GameModel project . . . 65
  - 3.3.1 DrawingStructure directory . . . 66
  - 3.3.2 DrawingAlgorithms directory . . . 68
  - 3.3.3 CanonicStructure directory . . . 69
  - 3.3.4 Solver directory . . . 71
  - 3.3.5 Synchronization of the representations . . . 73
- 3.4 NetworkClasses project . . . 74
  - 3.4.1 Project components . . . 74
  - 3.4.2 Communication protocol . . . 75
- 3.5 Server project . . . 79

3.5.1 Project components 79
3.5.2 Architecture of the server 80
3.6 UserInterface project 81
3.6.1 Auxiliary classes 82
3.6.2 Graphical controllers 83

4 User guide **85**
4.1 Sprouts: A Drawing Game 85
4.1.1 Installation 85
4.1.2 Tabs and Options menu 85
4.1.3 How to play 86
4.1.4 Quick play mode 87
4.1.5 Campaign mode 88
4.1.6 Remote game mode 89
4.1.7 Custom maps 91
4.2 Server application 92

5 Conclusion **93**
5.1 Achieved goals 93
5.2 Future work 94

Bibliography **95**

A Attachments **98**

1. Introduction

Sprouts[1] is a popular combinatorial paper-and-pencil game for two players with very simple rules:

- The game starts with n initial spots for some positive integer n.
- Players alternate in connecting spots by curves.
- After a player draws a curve, he also adds a new spot along that curve.
- No curve can cross or touch another curve or itself.
- Each spot can be incident to at most three curves.
- The first player who cannot make a move loses the game.

We can see an example of a Sprouts game with 2 initial spots in Figure 1.1. The first player loses the game there because it is his turn after the four curves have been made and there is no remaining valid move.

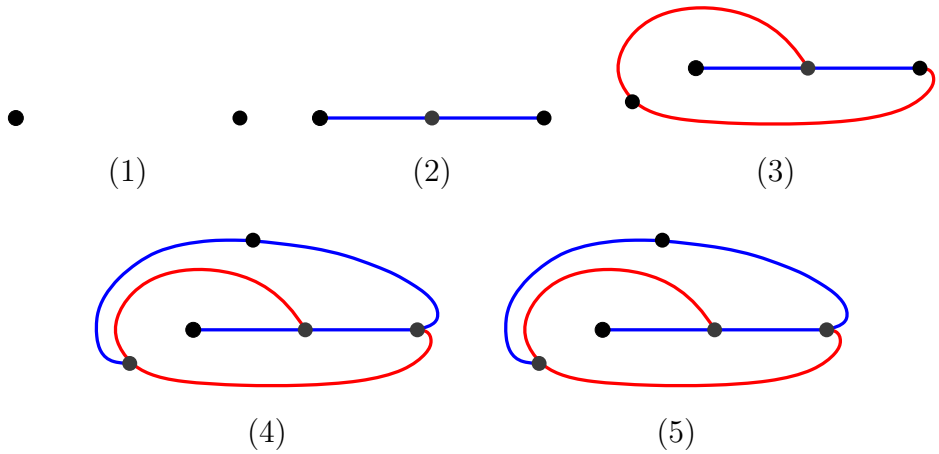


Figure 1.1: An example of a 2-spot game.

Sprouts was invented by British mathematicians John Horton Conway and Mike Paterson in 1967 with an intention to create a game that would resist to a computer analysis [2, 3]. Sprouts became increasingly popular over the years and eventually first books and articles analyzing the game were published. Nowadays, we can find dozens of publications about Sprouts; see [4]. Yet, even after more than half a century, we can hardly find a solid implementation of Sprouts that would allow us to play against a computer opponent [5]. This is quite surprising, given how simple-looking and famous the Game of Sprouts is. Moreover, as we will see later, many tools analyzing Sprouts positions are available [6]. Therefore, it is natural to wonder why there are essentially no Sprouts AI players.

We identify three main implementation barriers that cause many difficulties to potential developers. The first such a difficulty in the Game of Sprouts is to handle free-form input drawings that represent the positions. Additionally, the positions

[1] In Czech, Sprouts is also known as *Výhonky* or *Šprouti* [1].

tend to degenerate and become confusing over time. So the developer has to come up with some methods how to maintain them stable and clear. The second obstacle is that the number of possible moves can be exponential with respect to the number of initial spots. Thus, the developer needs to handle exceptionally large game trees even for relatively small games. In particular, this makes it very difficult to create a solid computer opponent capable of playing well on games with a higher number of spots. On top of that, it is highly nontrivial to synchronize the free-form inputs of a human player with the game representation used by a computer opponent.

The goal of this thesis is to deal with all these challenges and to create the first user-friendly Sprouts implementation with a strong artificial intelligence. To design a strong artificial intelligence we apply techniques used by the state-of-the-art Sprouts solver created by Lemoine and Viennot [6]. The extended abstract of this thesis is going to appear in the Proceedings of 29th International Symposium on Graph Drawing and Network Visualization (Graph Drawing 2021).

1.1 History of Sprouts

On February 21 1967, at the University of Cambridge mathematicians John Horton Conway and Mike Paterson invented an original paper-and-pencil game called Sprouts. It was the result of their intention to create a game that is simple to play and yet hard to analyze [2, 3].

Sprouts was later popularized by Martin Gardner [7] with his *Mathematical Games* column in the July 1967 issue of the American popular science magazine *Scientific American*. In 1969, Piers Anthony mentioned the Game of Sprouts in his science-fiction novel *Macroscope* [8] where one of the characters says, "Sprouts is an intellectual game that has had an underground popularity with scientists for a number of years." Few years later, David Pritchard, a British chess player and an editor of the magazine *The Gamer*, listed Sprouts in his compilation of world's best games for two players [9].

Shortly after the Game of Sprouts was invented, researchers tried to determine which player has a winning strategy in a game with n spots. It has been known for a long time that the second player has a winning strategy in the 1-spot and 2-spot games while the first player is winning in 3-spot, 4-spot, and 5-spot games. Later, Denis P. Mollison [10], a Cambridge mathematics student, proved that the second player has a winning strategy in the 6-spot game. His proof was made by hand and it took 49 pages; see Figure 1.2 for a full game tree of the 2-spot game. Thanks to this proof, Mollison won a 10-schilling bet with Conway that he cannot make a complete proof of the 6-spot game within a month [10]. Several years later, in 2001, Focardi and Luccio [11] proved by hand that the second player has a winning strategy in the 7-spot game using a relatively simple and clever case analysis.

In 1991, Applegate, Jacobson, and Sleator [12] published the first computer analysis of Sprouts and they successfully determined the winning player of n-spot games for each $n \leq 11$. They also introduced the famous *Sprouts conjecture* (Conjecture 4) which states that each n-spot game is winning for the first player if and only if n is equal to 3, 4 or 5 modulo 6.

The best results so far are due to Lemoine and Viennot [6] who created the

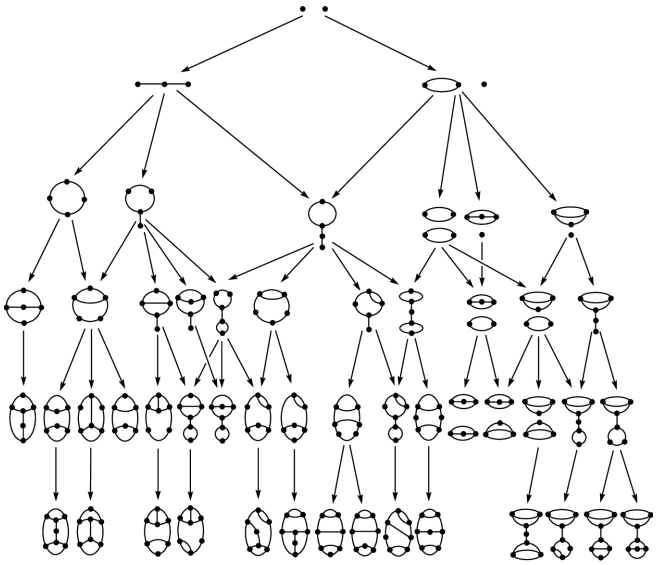


Figure 1.2: A full game tree of the 2-spot game (taken from [12]).

program *GLOP* that is able to determine the winning player of an arbitrary Sprouts position. In 2007, they were able to compute outcomes of all n-spot games for $n \leq 32$ and later, in February 2011, they improved their solver and determined all the outcomes up to 44 spots and even some outcomes up to 53 spots [13]. Each of the computed outcomes agrees with the Sprouts conjecture, which, however, still remains open.

Since 2001, there is an active *World Game Of Sprouts Association*, which registers Sprouts players and it regularly held Sprouts championships [14]. [2]

1.2 Theory behind Sprouts

In this section we will show basic properties of Sprouts. From these properties, we will derive a basic strategy for playing this game. First, we need to introduce a few definitions and auxiliary results.

1.2.1 Terminology

Here we formally describe a position in the Game of Sprouts and we introduce some notation that is also used later in this thesis. We also recall some basic terms from graph theory.

Let G be a graph of maximum degree at most 3. The number of *lives* of a vertex v in G, denoted by $l(v)$, equals $3 - \deg(v)$, where $\deg(v)$ is the degree of v in G. A vertex v with $l(v) > 0$ is called *alive*, otherwise it is *dead*. An *arc* is the image $\gamma([0,1])$ of an injective continuous mapping $\gamma\colon [0,1] \to \mathbb{R}^2$.

Let $\mathcal{G}$ be a class of plane graphs with maximum degree at most three that can be obtained from a finite set of isolated vertices by a sequence of the following *moves*: if u and v are (not necessarily distinct) vertices of a plane graph G from $\mathcal{G}$ lying in the face F of G and satisfying $l(v) + l(u) \geq 2$ and $l(u), l(v) \geq 1$, then we

[2] There is also a great YouTube video about Sprouts posted on *Vsauce2* channel. At the beginning of 2021, it reached more than 2 million views [15].

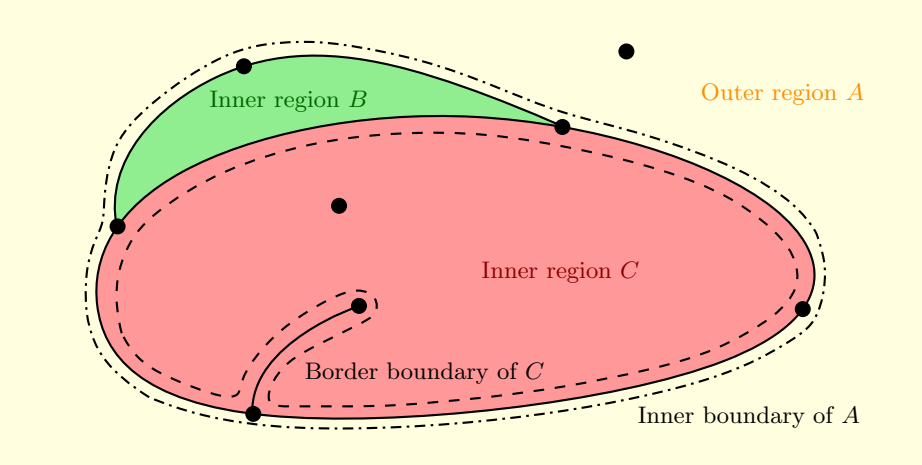


Figure 1.3: A Sprouts position with three regions and with an example of a border boundary and an inner boundary.

place a new vertex w inside F and we connect u with w and w with v by two interior-disjoint arcs contained in F.

A *position* in the Game of Sprouts is a plane graph from $\mathcal{G}$. Each vertex of this graph corresponds to one of the initial spots or to one of the spots added along the drawn curves. Edges of this plane graph are formed by the portions of the curves between two spots. For a positive integer n, the *n-spot position* is the position where the plane graph consists of n isolated vertices.

The faces of a position P are called *regions*; see Figure 1.3. The outer face of P is called the *outer region* of P and is denoted by R_O. We call all the other faces *inner regions* of P. For a region R of a position P, we define P_R as the plane subgraph of P induced by vertices that are incident to R. Every connected component of P_R is called a *boundary* of R. For an inner region R, the boundary that is incident to the outer region of P_R is called the *border boundary* of R. Note that the border boundary is unique. The other boundaries are called *inner boundaries*, including all boundaries of the outer region. The number of lives of a region R, denoted by $l(R)$, is equal to $\sum_{v \in R} l(v)$, where $v \in R$ denotes that v is incident to R. If $l(R) > 1$, then the region R is *alive* and it is *dead* otherwise.

The number of lives of a position P, denoted by $l(P)$, is equal to $\sum_v l(v)$ where the summation goes over all vertices v of P. A position P is *exhausted* if each region of P is dead. Equivalently, P is exhausted if and only if there is no valid move in P.

1.2.2 Basic properties

The first result is an important step used for deriving a basic strategy how to play Sprouts. First, we need to define a *survivor* as an alive vertex in an exhausted position. Note that each survivor has exactly one life, as otherwise it would be possible to make a loop from it as a valid move.

Proposition 1. *For every positive integer k, if a position P with $l(P) = k$ becomes exhausted after m moves, then there remain exactly $k - m$ survivors.*

Proof. Each move consumes two lives for the vertices that are being connected by

a curve. The vertex added along that curves retrieves one life. Thus, each move consumes one life in total.

It follows that if the position P becomes exhausted after m moves, then there remain $k - m$ lives. Since each survivor has exactly one life and there is no other type of alive vertices, there remain exactly $k - m$ survivors. □

Now we will prove that each Sprouts game ends after a finite number of moves.

Proposition 2. *For every positive integer k, each position P with $l(P) = k$ becomes exhausted after at most $k - 1$ moves.*

Proof. Since P has k lives and each move consumes exactly one life, we know that the position P becomes exhausted after a finite number m of moves. We will show that $m \leq k - 1$.

Let s be the number of survivors in the exhausted position reached from P. By Proposition 1, we obtain the equality $s = k - m$. Further, we know that $s \geq 1$ since at least the vertex added along the last curve remains alive. Thus, we get $1 \leq k - m$, which can be rewritten as $m \leq k - 1$. □

Sprouts is a type of game that Conway calls an *impartial game* [16]. These are games in which both players have the same possible moves in each position. Since Sprouts is also a finite by Proposition 2, we obtain the following result.

Observation 3. *For each position there exists a winning strategy either for the first or for the second player.* □

Currently, it is not known which positions in general are winning for the first player and which are not. Nevertheless, Applegate, Jacobson, and Sleator [12] stated the following conjecture, which was verified for each n-spot position computed so far.

Conjecture 4 (Sprouts conjecture [12])**.** *For every positive integer n, the first player has a winning strategy in the n-spot position if n is equal to 3, 4 or 5 modulo 6 and the second player is winning otherwise.*

The following statement is a direct corollary of Proposition 2 since each initial spot has exactly three lives.

Corollary 5. *For every positive integer n, the n-spot position becomes exhausted after at most $3n - 1$ moves.* □

Before proving the lower bound on the number of moves in the n-spot games, we define two additional types of vertices occurring in an exhausted position. The first such type of vertices is called a *neighbor*. Neighbors are vertices that are incident to a survivor. If a survivor is incident to only a single vertex v, then the other vertex incident to v (besides the survivor itself) is also called a neighbor. These two types of neighbours are shown in Figure 1.4. All the remaining vertices in an exhausted position are called *pharisees* (from the Hebrew for "separated ones"). Note that each survivor has exactly two neighbors.

Proposition 6. *For every positive integer n, it takes at least $2n$ moves before the n-spot position becomes exhausted.*

Two adjacent neighbors. One adjacent and one distant neighbor.

Figure 1.4: The two types of neighbors of the black vertex.

Proof. Let P' be an exhausted position reached from the n-spot position P after m moves. Since each move adds a new vertex, then the exhausted position P' contains exactly $n+m$ vertices. The position P' consists of s survivors, b neighbors, and p pharisees. Therefore we get

$$n + m = s + b + p.$$

We show that each neighbor is a neighbor of only one survivor. Suppose for contradiction that there is a neighbor of at least two survivors. Then, the survivors are in the same region and hence it is possible to make a move between them which contradicts the assumption that P' is exhausted. Thus, since each survivor has exactly two neighbors, we obtain $b = 2s$. By Proposition 1, we get $s = 3n - m$.

Substituting for n and b in the previous equality and rearranging it further gives

$$\begin{aligned} n + m &= (3n - m) + 2(3n - m) + p \\ m &= 2n + p/4. \end{aligned}$$

Since $p \geq 0$, we obtain the desired inequality $m \geq 2n$. □

Both the upper bound from Corollary 5 and the lower bound from Proposition 6 are tight, as shown in Figure 1.5 and Figure 1.6, respectively.

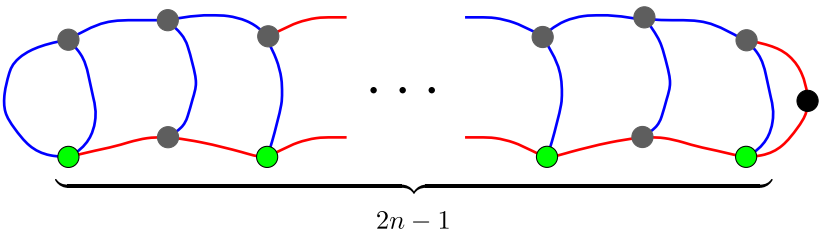


Figure 1.5: An example of a Sprouts position with n initial spots and $3n-1$ moves.

1.2.3 Game strategy

By Proposition 3, there exists a winning strategy either for the first or the second player at any position. Even though no such strategy is known in general, there are a few basic strategies that can improve our game. The most important one is based on the following result.

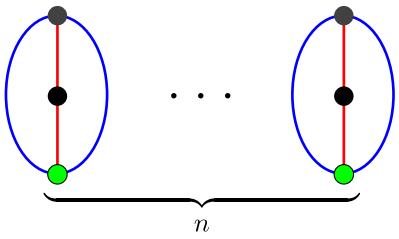


Figure 1.6: An example of a Sprouts position with n initial spots and $2n$ moves.

Proposition 7. *Let P be a position with $l(P) = k$ and let P' be an exhausted position with s survivors that is reached from P. Then, the player on the move at P wins the game at P' if and only if the parity of k and s is different.*

Proof. ($\Longrightarrow$) If the starting player wins the game at P', then there is an odd number m of moves between P and P'. It follows from Proposition 1 that $m = k-s$. Since m is odd, the parity of k and s has to be different.

($\Longleftarrow$) We will prove the following implication: if the first player loses at P', then k and s have the same parity. Losing as the starting player at the position P implies that the game took an even number m of moves. Analogously as before, it follows from Proposition 1 that $m = k - s$. Since m is even, the parity of k and s has to be same. □

In other words, Proposition 7 states that the starting player at a position with an even (odd, respectively) number of lives needs to finish the game with an odd (even, respectively) number of survivors. Together with this strategy there are a few tricks how to affect the final number of survivors. For example, in the n-spot position the game could end with anywhere from one up to n survivors. However, if a player makes a loop in such a position with $n \geq 3$ and separates the other spots so that in each of the newly created regions will be at least one spot, then he increased the minimum number of survivors at the end of the game by one. The reason is that at least one survivor will remain in both of the regions and it will not be possible to connect them.

1.2.4 Sprouts variations

There are many variations of Sprouts in the literature and here we would like to draw attention to two of them.

The first variation is called *Misère Sprouts* [17] and it has almost the same rules as the original version, the only difference is that the first player who cannot make a move *wins* the game. Perhaps surprisingly, analyzing positions of Misère Sprouts is much more difficult than analyzing positions of the standard variation. While outcomes of the n-spot games in the standard variation are computed up to 44 spots, the outcomes of Misère Sprouts are known only up to 20 spots [13].

Another Sprouts variation is called *Brussels Sprouts* [18]. Instead of n initial spots the game starts with n initial crosses with four free ends. Similarly to the standard Sprouts, players alternate in drawing curves between the free ends of the crosses. After each curve is drawn, a new cross with two free ends is made by adding a stroke across that curve; see Figure 1.7. Also in the Brussels Sprouts the curves cannot cross each other and also the first player without a possible

move loses the game. One can prove, using Euler characteristic for connected plane graphs, that the game with n initial crosses always ends after exactly $5n - 2$ moves [1]. It follows that the starting player in games with an odd number of initial crosses always wins the game and the opponent cannot do anything about it. Analogously, the starting player always loses in games with an even number of initial crosses.

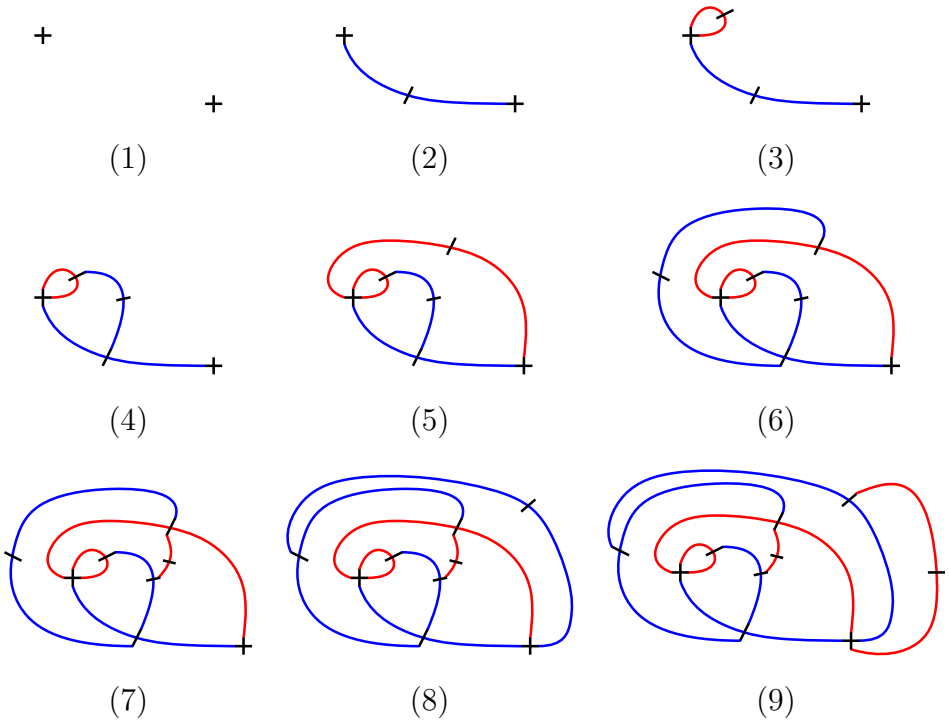


Figure 1.7: An example of a Brussels Sprouts game with 2 initial crosses.

1.3 Existing implementations

As already mentioned, we lack Sprouts implementations with a computer opponent [5]. In this section, we list all the Sprouts applications that we are aware of. This list mostly includes implementations that allow to play only against human players, as we have found only a single program that allows play against AI players.

We remark that besides the listed implementations there were some intentions to create Sprouts applications with AI [5, 19, 20]. For example, in his paper from 2016, Browne [5] mentioned creating a complete Sprouts-playing app and investigating AI methods for playing the game at arbitrary sizes. However, no such result has been published yet.

Sprouts - A Game of Maths! This is a Sprouts implementation released by Gehrig [21, 22] in 2011 and it was written to run on iPads.

The game was mentioned in the paper by Browne [5] as one of two representatives of Sprouts implementations that support a game against AI players.

However, he states that the application did not work on any tested device, which later probably resulted in the game being removed from the App Store. Since that it is unavailable. As noted by Browne [5], the game supposedly supported games on up to 15 spots. However, as we can see in Figure 1.8, the game allows degenerated positions with almost touching curves.

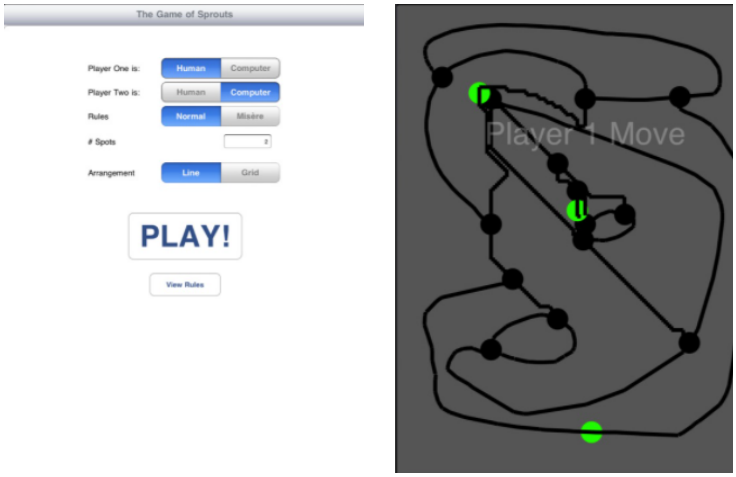


Figure 1.8: Screenshots of *Sprouts - A Game of Maths!*.

SproutsPlus The game was released by Christian E. Altmann [23] on the App Store in 2019. The name of this game is sometimes spelled as *Sprouts++*.

This game can be hardly marked as a Sprouts implementation as the rules differ. The application does not forbid mutually crossing curves, it does not detect the end of the game, and, furthermore, it does not allow to make some kinds of moves (at least the attached instructions do not describe how to make loops); see Figure 1.9 for an example of crossing curves. Another discomfort for a potential user is that the implementation does not allow to draw the curves freely. The user has to mark two spots that should be connected and the game then makes a curve between them. This curve can be adjusted only by changing the position of the new newly placed spot. The game does not include a computer opponent nor a tool to keep the positions clear.

Among positive features of the implementation, we can include the fact that the game contains a certain type of a remote game and it seems to allow games with an arbitrary number of spots.

Sprouts Game This Sprouts implementation was created by Gokhan Egri [24] in 2019 and it is another Sprouts application designed for iOS.

Sprouts Game is a simple implementation that mostly just simulates the game of two players on a sheet of paper. The game offers only one additional feature in comparison with playing on paper and that is displaying the number of remaining lives on each spot. Unlike SproutsPlus, Sprouts Game supports freely drawn curves and there is no type of Sprouts moves that cannot be made. However, similarly to SproutsPlus, it allows mutually crossing curves; see Figure 1.10. Therefore the program is also not able to determine the end of the game properly. Furthermore, it does not redraw game positions, and thus not prevent them from degenerating.

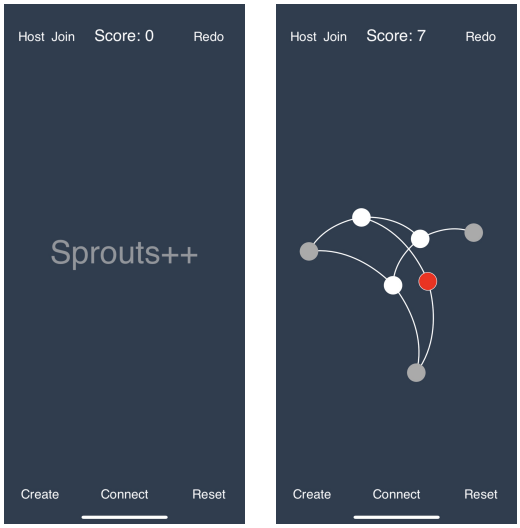


Figure 1.9: Screenshots of SproutsPlus.

The application supports only local games and it also does not contain any AI opponent.

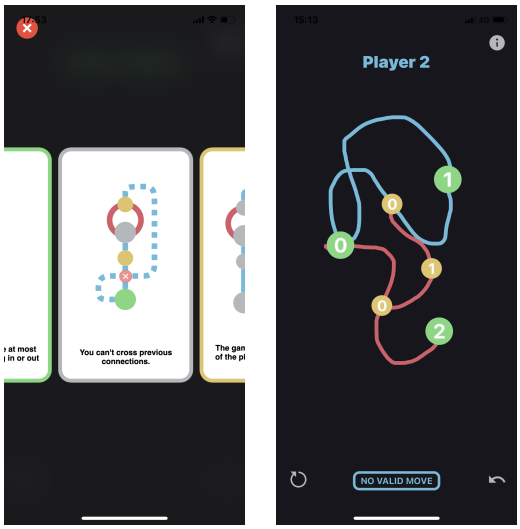


Figure 1.10: Screenshots of Sprouts Game.

University of Utah Sprouts Applet In the math department of the University of Utah, Peter Alfeld [19] developed a Sprouts applet allowing a game against a human player. It was created in 1999 as an application for Java compatible browsers.

The implementation and its user interface is very simple; see Figure 1.11. The application does not keep games clear, as the positions are fixed throughout the whole game. It allows only local games of two human players. Among positive features, the game prevents crossings of the curves and it provides essentially unlimited size of the game.

On the website of the project [19], the author states that the applet will eventually offer a game against AI players because he is intrigued with the

project, as it involves fascinating programming problems. However, the applet has remained without any AI player until today.

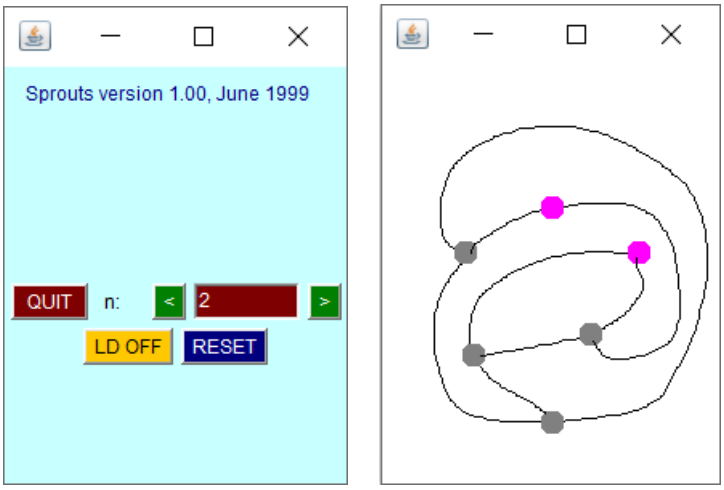


Figure 1.11: Screenshots of University of Utah Sprouts Applet.

3Graph In 2009, Stefan Reiss [25] created a Sprouts implementation called 3Graph. It was designed for Windows and, in our opinion, it is the best implementation of Sprouts that has been created so far. The game was released and described on the author's website [25].

In 3Graph, a player can connect two spots either by freely drawn curves or by marking two spots that should be connected and selecting a move among all possible types of connections. In comparison with the previous implementations, both of these input options prevent crossings. Furthermore, 3Graph is the only implementation that came up with a method of maintaining Sprouts positions stable and clear. This is done by a redrawing algorithm that moves the curves after each move so that the distance between each curve is sufficient to place another move there. In Figure 1.12, there is an example of a Sprouts position after eight moves have been done and, as we can see, the position is nicely spread out and there is no pair of curves that would be too close to each other.

The application contains a campaign that consists of 24 levels. On each level there is a randomly generated Sprouts position that can be obtained from an n-spot position with m moves, where $n \leq 8$ and $0 \leq m \leq 5$. The game 3Graph supports a computer player that plays perfectly on each position in the campaign. It is also possible to generate a custom game with up to 8 spots and up to 5 initial moves. The player can then play on this generated position locally against a human player or against different levels of AI. However, it is not guaranteed that the most advanced AI plays perfectly on each such a position.

Even though 3Graph is the best Sprouts implementation created so far, there are several places for improvement. The first drawback is its user interface. The graphics of 3Graph is very simple and a lot of features of the game are not intuitive. Furthermore, as the player gradually passes through the campaign to the positions with higher number of spots, the game starts suffering with various errors. The game often crashed due to internal errors and, quite rarely, the curves can even cross each other, which results in another fatal internal error. 3Graph does not

	Currently available	Free-form input	Crossings detection	Maintaining positions	Computer opponent	Remote game	Target platform
Sprouts - A Game of Maths!	✗	?	?	?	✓	?	iOS
SproutsPlus	✓	✗	✗	✗	✗	✓	iOS
Sprouts Game	✓	✓	✗	✗	✗	✗	iOS
UoU Sprouts Applet	✓	✓	✓	✗	✗	✗	Applet
3Graph	✓	✓	✓	✓	✓	✗	Windows

Table 1.1: An overview of the existing Sprouts implementations.

allow games with a higher number of initial spots, which is a pity because such large games can be interesting to play even if there was not a perfectly playing AI. And finally, it would be nice if such a high-quality implementation of this famous game offered the option to play remotely.

Figure 1.12: Screenshots of 3Graph.

Summary In Table 1.1, we can see an overview of the currently existing Sprouts implementations. We summarize all the main features that, in our opinion, a high-quality implementation of Sprouts should support. We also mark the availability of these features in all the listed applications.

1.4 Our goals

As we have seen in the previous section, 3Graph is the only solid implementation of Sprouts currently available. However, even this game can be significantly improved. Thus, the goal of this thesis is to create the first user-friendly Sprouts implementation with a strong artificial intelligence and with a support of a remote game. In this section, we summarize all the features that our implementation should support.

G1 *Free-form input curves*: it should be possible to insert curves simply by drawing them. Drawing should be comfortable and it should enforce the

user to follow the Sprouts rules. In particular, the player should not be allowed to make crossing curves.

G2 *Maintaining positions clear*: the application should include a redrawing tool that will maintain the positions stable and clear. The algorithm should be fast enough so that it could be computed in real-time, and thus does not slow down the game too much.

G3 *Strong artificial intelligence*: the implementation should contain various levels of a computer opponent that should be able to play on each position obtained from the n-spot positions with n up to 15. The most advanced one should be able to play perfectly on pre-trained positions obtained from the n-spot positions with n up to 8 and possibly some with $n > 8$.

G4 *User-friendly GUI*: the user interface should be simple enough so that using the application will be highly intuitive even for players who have never played Sprouts before. This includes clear menus and a user guide describing the application.

G5 *Remote game*: besides local games against a human or a computer opponent, the players should be able to connect to a Sprouts server that would make it possible to create game rooms and play against other players online.

G6 *Custom games*: the user should be able to create and store his own positions and play on them locally against another human or a computer player or remotely against other human players.

2. Analysis

In the preceding chapter, we identified three main implementation barriers of the Game of Sprouts. In this chapter, we will discuss them in more detail and propose their solutions.

The first difficulty is to handle free-form input drawings that we decided to support in Goal **G1**. In Section 2.1, we design a data structure that can store these positions and allows us to perform the necessary operations on them effectively. One problem with these drawings is that the positions tend to degenerate and become confusing during a game. To keep the positions stable, we modify a force-directed algorithm called *ImPrEd* that was introduced by Simonetto, Archambault, Auber, and Bourqui [26]. Our modification, which is described in Section 2.2, turned out to be suitable for Sprouts and keeps the game clear and comfortable, which satisfies our Goal **G2**.

In Goal **G3**, we decided to implement a strong computer player. To reach this goal, we have to deal with the two remaining implementation barriers. The first one is possibly exponential growth rate of the number of moves with respect to the number of initial spots, which results in very large game trees. To handle these trees, we use an idea that was inspired by the already mentioned program GLOP [6]. This is currently the best Sprouts solver and it is based on the *Alpha-beta pruning algorithm*. Since Sprouts positions often consist of many independent sub-positions, the authors of GLOP applied the theory of *Grundy numbers*, also called *nimbers*, to speed up the computation of the outcome of a position. This is done by computing the nimber of each of the sub-positions separately, merging them, and, based on the result, determining the outcome of the whole position. Although this improvement highly increases the performance of the computation, it is still not enough to cover positions with a higher number of spots. Luckily, many moves lead to mutually equivalent positions that can be identified and merged together by a proper encoding of Sprouts positions. These methods together with our database of pre-trained positions allow us to create a strong computer opponent who can play perfectly on larger positions than any other Sprouts implementation with AI does. All the techniques that we used to design the AI players are described in Section 2.3.

After a computer move is determined, we need to be able to automatically draw it in an arbitrarily Sprouts position. However, it is highly nontrivial to synchronize the representation of a move used by a computer opponent with the graphical representation. This is the final main issue we have to deal with. To solve this, we come up with our own solution described in Section 2.4. It is based on a triangulation of the plane and a method that attaches freely lying inner boundaries to their border.

2.1 Graphical representation

To handle Sprouts positions, we need to design an efficient data structure. Many researchers analyzing Sprouts [5, 6, 12] use a so-called *string representation*, where each vertex is denoted by a letter and the resulting position is represented as a certain sequence of these letters. This approach is strong enough to express

game properties of arbitrary Sprouts positions and allows us to analyze them. In Section 2.3, we also use this data structure to implement a computer player.

However, the string representation is not strong enough to fully capture the graphical properties of positions that are drawn in our program. Therefore, we extend the string representation into a *graphical representation* that fully captures the shape and the game properties of any Sprouts position. In particular, this representation suffices for any graphical algorithm. Another advantage is that, since the graphical representation is an extension of the string representation, the conversion from the graphical representation to the string representation is trivial. Since performing this conversion is efficient, we can retrieve the string representation from the graphical one whenever we need it.

2.1.1 Components of the graphical representation

Here, we will describe individual parts of the graphical representation. This includes representations of vertices, edges, boundaries, and regions of a Sprouts position.

Let P be a Sprouts position drawn in the game. We assume that the edges of P are drawn as piece-wise linear arcs. In the *graphical representation* $gr(P)$ of P, we represent an edge e of P by a sequence $gr(e)$ of vertices of the corresponding piece-wise linear arc. A boundary β of P is represented by a sequence $gr(\beta)$ of the sequences representing the edges of β. A region R is then represented by a set $gr(R)$ of the lists representing the boundaries of P organized into a tree-like hierarchical structure. We describe the representation $gr(P)$ in more detail in the rest of this subsection.

From now on, we will not explicitly distinguish between edges, boundaries, regions and their graphical representations unless it is necessary.

Vertices The building stones of our graphical representation $gr(P)$ are vertices. Vertices of $gr(P)$ are the endpoints of the line segments forming the piece-wise linear curves representing the edges of P or Sprouts spots that stand alone somewhere in a region of P. The vertices in the latter case are called *singletons*. We distinguish two types of vertices. The first type are *game vertices* that correspond to the real Sprouts spots. They are either singletons or they lie on one of the ends of the arcs that represent edges of P. All other vertices are *inner vertices*. We denote the graphical representation $gr(v)$ of vertex v by a small letter in general but we distinguish game vertices by capital letters. In Figure 2.1, there are 19 different game vertices, where two of them are singletons. Also the inner vertices of the edge j are displayed.

We also need to specify how to place the vertices in the plane. Stefan Reiss [25], the author of 3Graph, placed vertices in a grid of a gradually increasing size. We suspect that this approach was chosen because the number of potential placements of a vertex is highly limited, and thus a redrawing algorithm might be more efficient. However, in our opinion, this method leads to a rather large case analysis and there exist other methods how to obtain an efficient redrawing algorithm; see Section 2.2.

Therefore, we decided not to place vertices on a grid, but we allow the vertices to lie anywhere in the square $[0, 1]^2$. Limiting the positions of vertices to the

square $[0,1]^2$ allows us to draw positions on displays of arbitrary sizes without any complicated conversions. This is used, for example, in the remote game, where we can share the same representation of a position and both devices can easily scale the drawing based on the size of their display. The arbitrary placement of the vertices results in drawings that are finer than in 3Graph.

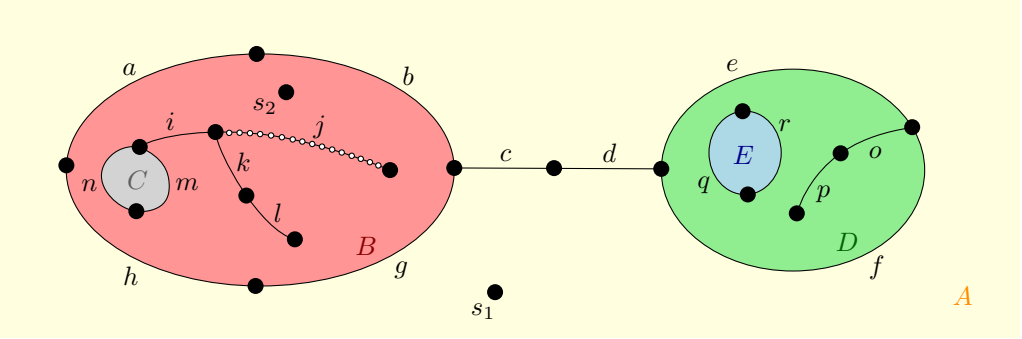


Figure 2.1: A position with edges labelled a–r and with regions labelled A–E.

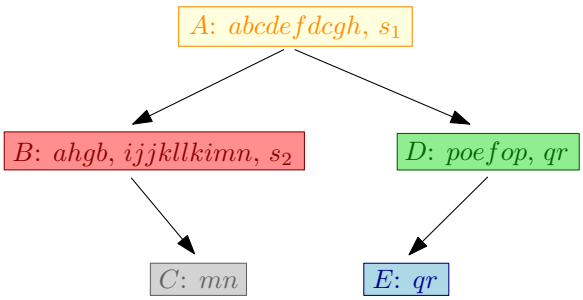


Figure 2.2: The graphical representation of the position from Figure 2.1, where s_1 and s_2 denote singletons.

Edges The next structure in our graphical representation are edges. An edge e of the position P is represented by a sequence $gr(e) = (S, i_1, \ldots, i_n, E)$ of vertices, where the *starting vertex* S and the *ending vertex* E are game vertices corresponding to the endpoints of e and $i_1, \ldots, i_n$ are the inner vertices of e. The ordering of $gr(e)$ is determined by the order in which the edge e was drawn. Any two consecutive vertices of $gr(e)$ are connected by a line segment that we call a *small edge*. In Figure 2.1, we can see 18 different edges of P. The inner vertices of the edge j are highlighted. Even though the edges are represented as pieces-wise linear arcs, the edges are displayed as smooth curves in the actual game using interpolation by *Bézier curves*.

Boundaries Other structure that we store in the graphical representation are boundaries. A boundary β is represented by a sequence $gr(\beta) = gr(e_1), \ldots, gr(e_n)$ of sequences $gr(e)$ for all edges e of β in the order we meet them as we traverse β. All the inner boundaries in a region of P are oriented clockwise. The border boundary is oriented counterclockwise. In order for our edge-insertion algorithms to work properly, it is necessary that all inner boundaries of a region have the

same orientation while the orientation of the border boundary is opposite. The edge-insertion algorithms are described in Subsection 2.1.3.

In the text, we denote the boundaries as follows. The sequence $gr(\beta) = gr(e_1), \ldots, gr(e_n)$ is denoted by a string $e_1 \cdots e_n$ called the *edge-sequence*. We sometimes simplify an edge-sequence $e_1 \cdots e_n$ to a *vertex-sequence* $G_1 \cdots G_n$, where G_i is the starting vertex of $gr(e_i)$ for $i = 1, \ldots, n$. Each term e_i is called an *edge occurrence* and each term G_i is called a *vertex occurrence*. Note that it does not matter which rotation of $gr(\beta)$ we consider. For example, we could use the edge-sequence $e_2 e_3 \cdots e_n e_1$.

In Figure 2.1, the region B contains three boundaries. The border boundary of B has the edge-sequence sequence $ahgb$. The edge-sequence $ijjkllkimn$ represents the non-singleton inner boundary of B. The last inner boundary of B contains only a singleton vertex and no edge. We represent such cases by a single edge that has no inner vertices and whose starting and ending vertex is the singleton. Note that the border boundary of B has the counterclockwise orientation and the non-singleton boundary of B has the clockwise orientation.

Since the orientation of each occurrence of an edge e in $gr(\beta)$ has to correspond with the orientation of $gr(\beta)$, we sometimes need to reverse a certain occurrence of $gr(e)$ in $gr(\beta)$. For example, the edge l in Figure 2.1 occurs twice in the edge-sequence $ijjkllkimn$ of its boundary γ. Since $gr(\gamma)$ is oriented clockwise, the first occurrence of l in $gr(\gamma)$ should have the opposite orientation than the second occurrence of l in $gr(\gamma)$. This problem does not appear only on a single boundary. For example, the edge a in Figure 2.1 must also have the opposite orientation in the boundary $ahgb$ than the occurrence of a in the boundary $abcdefdcgh$.

This problem is solved by marking occurrences of each sequence $gr(e)$ in $gr(\beta)$ that need to be reversed with respect to the original orientation of $gr(e)$. The reversed occurrence of an edge e is denoted by e^R. For simplicity of notation, we will often not explicitly mark occurrences that need to be reversed.

Regions The final structure in our graphical representation $gr(P)$ are regions. Every region R of P is represented by a set $gr(R)$ of the sequences $gr(\beta)$, where β is a boundary of R.

The regions of P define the area in which a new curve can be drawn, and thus they can be used to speed up drawing of a new move, as we only need to check the intersections of the curve currently being drawn with the edges of the corresponding region. We also use regions to mark survivors that can no longer affect the game and also to make our redrawing algorithm from Section 2.2 more efficient.

We recall that every inner region R of P has a unique border boundary. Since the edges of P correspond to piece-wise linear arcs, the border boundary determines a (not necessarily simple) polygon that we call the *border polygon* of R.

To identify regions that contain a given point more easily, we organize the regions of P in the following tree-like hierarchical structure. The root of the tree is always the outer region of P. All the other regions of P are descendants of the outer region of P. An inner region R of P is a descendant of another region S of P if the whole border polygon of R is contained in the border polygon of S. If a region S is an immediate successor of a region R, then we call S a *subregion* of R.

Since our border polygons are not necessarily simple, the hierarchy of regions can be a little counter-intuitive sometimes; see Figure 2.3.

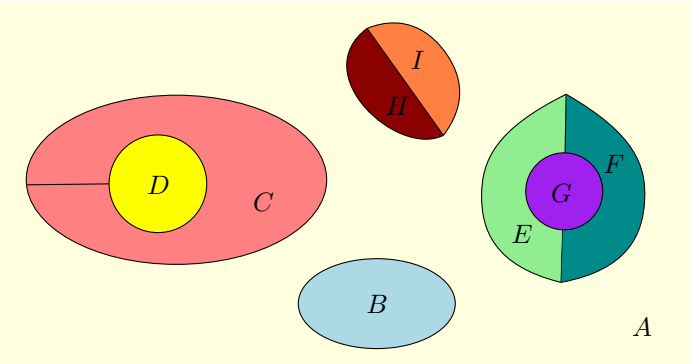


Figure 2.3: A position where each region is a subregion of the region A.

This concludes the description of all the components of the graphical representation. See Figure 2.2 for the graphical representation of the position from Figure 2.1.

2.1.2 Drawing a move

Now, we introduce first algorithms that work with the graphical representation. Specifically, we describe algorithms that allow us to draw moves with a crossing detection. Drawing of a move proceeds by drawing a sequence of the individual small edges of the resulting edge that have an optimal length l_{opt}. The length of the small edges can be later adjusted by the redrawing algorithm described in Section 2.2.

Point-polygon incidences Throughout the program, we often need to determine whether a point p is inside a polygon P, which is in most cases obtained from a boundary. The algorithm works with a horizontal ray r that starts at p and then moves indefinitely to the right. The point p lies inside P if and only if the number of intersections between the boundary of P and r is odd. Note that some edges can occur twice within a polygon. Then, the intersection with such an edge is counted twice; see Figure 2.4.

Searching for a region with a given point Our graphical algorithms often work locally within a region that is determined by a certain point that lies in it. Due to the hierarchical structure of regions, we can achieve this by using Algorithm 1 that goes through the descendants of a given region R and looks for a region that contains a given point p. To find a region in the graphical representation that contains a point p, we call the function $FindRegion(p)$.

Detecting crossings To reach Goal G1, we need to prevent crossings. To avoid them, we detect crossings while a move is being drawn. The drawn edge e can either cross itself or edges within the same region R. To detect self-crossings, we only check the crossings of the last drawn small edge s of e with all the other small edges of e.

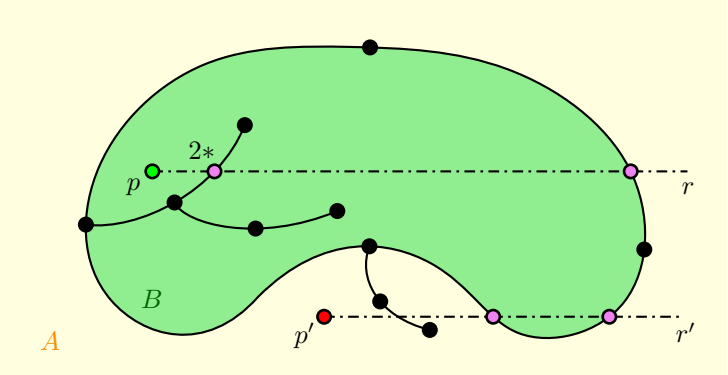


Figure 2.4: The algorithm that determines whether two given points p and p' lie inside the border polygon of the region B. The point p lies in B as r intersects the border polygon of B three times. The point p' lies outside of B as r' intersects the polygon twice.

Algorithm 1: Find a region with a given point p inside

```
Function FindRegion(p)
    O ← the outer region of the position P;
    return FindRegion(O, p);
Function FindRegion(R, p)
    P ← the border polygon of R;
    if p is inside P then
        foreach subregion S of R do
            T ← FindRegion(S, p);
            if T ≠ null then
                return T;
        return R;
    return null;
```

To detect crossings between s and other small edges in R, we use the function $FindRegion(p)$ from Algorithm 1 to find the region R where p is the last endpoint of the first small edge of e. Then, we only check crossings with the small edges within R.

2.1.3 Inserting a move

After a move is drawn, we would like to insert it into the graphical representation. There are two types of moves. Moves that connect two game vertices from different boundaries are called *double-boundary moves*, while moves that connect two (not necessarily different) game vertices from a single boundary are called *single-boundary moves*. Both types of moves are discussed in the following paragraphs along with the algorithms that insert them into the graphical representation.

Note that an edge e representing a drawn move is divided in the middle into two new edges e' and e'' since a new game vertex is added after each move. These two sequences are inserted into the graphical representation instead of the original

edge e. However, since our algorithms are the same for inserting e as for inserting e' and e'', the described algorithms will only insert the original edge e for brevity. We also assume that the vertices we are connecting have enough lives.

Double-boundary move Let $e = (S, i_1, \ldots, i_l, E)$ be the edge representing a drawn double-boundary move that connects boundaries α and β with the edge-sequences $f_1 \cdots f_m$ and $g_1 \cdots g_n$, respectively. The small edge Si_1 is connected to the occurrence S_j of the starting vertex of f_j in the corresponding vertex-sequence. Similarly, the small edge $i_l E$ is connected to the occurrence E_k of the starting vertex of g_k. Then, the double-boundary move between α and β creates a new boundary $f_1 \cdots f_{j-1} e g_k \cdots g_n g_1 \cdots g_{k-1} e^R f_j \cdots f_m$. In the case α is the border boundary of the region R in which the move is being made, we need to update the hierarchical structure of regions. We mark all subregions of R that are contained in the face bounded by β as subregions of the parent region of R and no longer as subregions of R. For example, consider the position from Figure 2.3 where the move is the edge connecting the border boundary α of the region $R = C$ with the border boundary β of the region D. We proceed analogously if β is the border boundary of R.

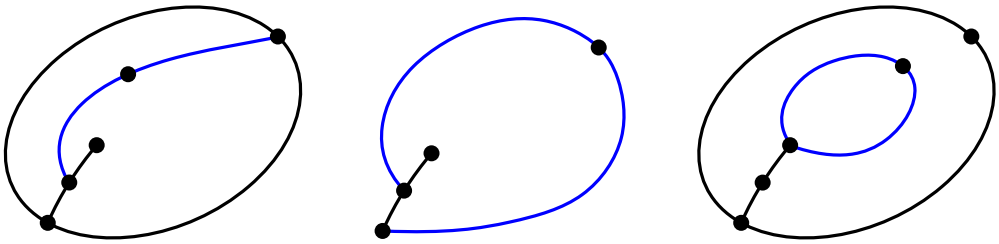
Single-boundary move Let $e = (S, i_1, \ldots, i_l, E)$ be the edge representing a drawn single-boundary move that connects an occurrence S_j of S and an occurrence E_k of E on a boundary α with the edge-sequence $f_1 \cdots f_j \cdots f_k \cdots f_m$ in a region R. If $S_j = E_k$, then we assume that the drawn loop has the counterclockwise orientation.

The single-boundary move on α creates two new boundaries β with the edge-sequence $f_j \cdots f_{k-1} e'$, called the *major boundary*, and γ with the edge-sequence $f_1 \cdots f_{j-1} e'' f_k \cdots f_m$, called the *minor boundary*, where each of e' and e'' is equal to either e or e^R. The actual values of e' and e'' are decided using a small case analysis that is shown in Figure 2.5 where six different types of single-boundary moves are listed.

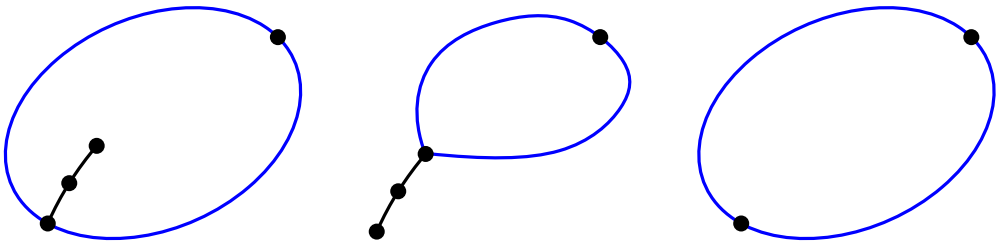
If α is an inner boundary, then a new region S is formed inside the region R. The border boundary of S is either β or γ. Since the move is made on an inner boundary, the orientations of β and γ are opposite. Thus, we can determine which of these two boundaries is the border boundary of S and which is the new inner boundary of R instead of the original boundary α. At the end, S becomes a subregion of R and all subregions of R together with all inner boundaries of R that lie in the border boundary of S are moved from R to S.

If α is the border boundary of R, then R is split into two new regions S and T. According to cases (a) and (c) from Figure 2.5, we see that both β and γ have the counterclockwise orientation and are the border boundaries of two newly formed regions S and T. All subregions and inner boundaries of R are partitioned between S and T. Finally, the region hierarchy is updated by removing the region R and by inserting the regions S and T as subregions of the parent region of R.

The region that contains the major boundary is called the *major region* and the region with the minor boundary is called the *minor region*. The set of boundaries in the major region except the major boundary itself is called the *major partition* and analogously for the *minor partition*.



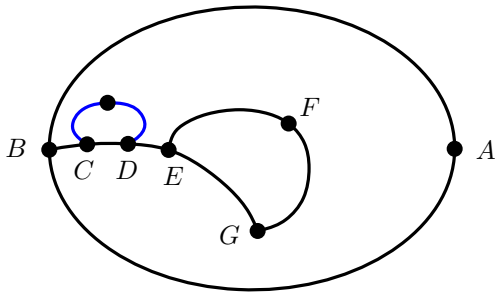
(a) $e' = e^R$, $e'' = e$ (b) $e' = e^R$, $e'' = e$ (c) $e' = e$, $e'' = e^R$



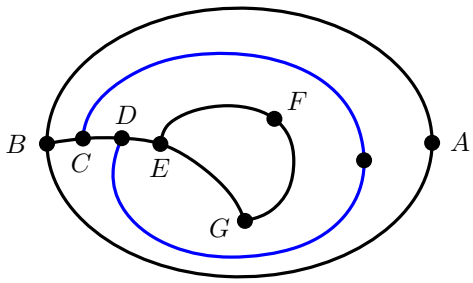
(d) $e' = e^R$, $e'' = e$ (e) $e' = e$, $e'' = e^R$ (f) $e' = e$, $e'' = e^R$

Figure 2.5: All types of single-boundary moves used to assign the values of e' and e''. The moves in parts (a)–(b) satisfy $S_j \neq E_k$ while we have $S_j = E_k$ in parts (c)–(f). In parts (a) and (c), the endpoints are on a border boundary and they are on an inner boundary in part (b), (d), and (e). In part (f), the boundary is a singleton. In part (d), all edges of the major boundary are contained in the minor boundary which does not hold in part (e).

Let e be an edge representing a drawn move. To insert e into the graphical representation, we need to correctly identify the occurrences of its endpoints in the boundary or the boundaries that e is attached to. It is important to correctly identify the occurrences as otherwise we might get a different position. See Figure 2.6 for two non-equivalent moves that differ only in the occurrences of vertices they connect.



Connecting the first occurrence of C and the first occurrence of D.



Connecting the first occurrence of C and the second occurrence of D.

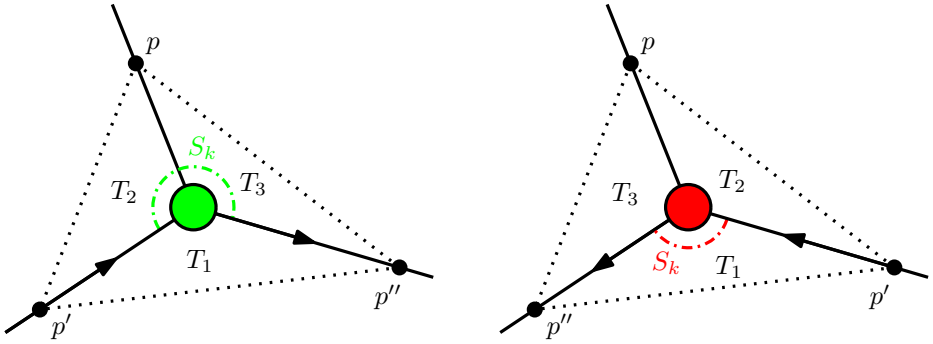
Figure 2.6: An example of two non-equivalent moves on a boundary with the vertex-sequence $ABCDEFGEDCB$. Note that both moves connect same pair of vertices.

Connecting a small edge to a vertex Consider boundaries α and β and an edge e connecting a game vertex S of α and a game vertex E of β. Since the vertices S and E may occur up to three times in the vertex-sequences of α and β, we need to identify the right occurrence of S and E for the correctness of our graphical algorithms. The information that our curve connects S and E is not sufficient to insert the move because we need to know exactly which occurrences of S and E are connected, which determines the sides from which the edge e approaches α and β.

By symmetry, it suffices to show how to correctly identify the occurrence of S in α. Let $e_1 \cdots e_n$ be the edge-sequence of α. Let Sp be the first small edge of e. The occurrence S_k of S in α is *the correct occurrence* if $\deg(S_k) \leq 1$ or if p lies on the left side of α with the respect to the orientation of α. Here, we are using the fact that our inner boundaries have the clockwise orientation and the border boundaries have the counterclockwise orientation.

See Figure 2.6 where the first move is correctly connected to the first occurrences of C and D and not to their second occurrences. Note that both correctly connected small edges are on the left side of the border boundary with the respect to its counterclockwise orientation. Connecting an occurrence of a vertex on an inner boundary is similar since the orientation of inner boundaries is clockwise.

We now show how to decide whether the point p lies on the left of α. We assume that S_k is the starting vertex of e_k. We take the last small edge $p'S_k$ of e_{k-1} and the first small edge S_kp'' of e_k and we consider three triangles $T_1 = p'S_kp''$, $T_2 = p'S_kp$, $T_3 = S_kp''p$; see Figure 2.7. Then, the position of p with respect to the the triple (p', S_k, p''), and thus the position of p with respect to to the boundary α, is determined by Table 2.1. In the table, we assume that $\deg(S_k) = 2$ and that the small edge S_kp is not collinear with $p'S_k$ nor with S_kp''.



The correct occurrence of S. The incorrect occurrence of S.

Figure 2.7: An illustration of the algorithm that checks if p lies on the left of α.

2.2 Redrawing algorithm

As mentioned before, Sprouts positions tend to degenerate, which often produces edges that are too close to each other or edges that are unnecessarily long. To avoid such edges and to reach Goal **G2**, we implemented a redrawing algorithm that is a modification of a simple force-directed algorithm called *PrEd* introduced

The orientation of T_1	The orientation of T_2	The orientation of T_3	The position of p to (p', S_k, p'')
counterclockwise	counterclockwise	counterclockwise	on the left
counterclockwise	both of them are not counterclockwise		on the right
clockwise	clockwise	clockwise	on the right
clockwise	both of them are not clockwise		on the left
collinear	counterclockwise	counterclockwise	on the left
collinear	clockwise	clockwise	on the right

Table 2.1: The position of p depending on the orientations of T_1, T_2, and T_3.

by Bertauls [27]. The algorithm PrEd was later improved to an algorithm *ImPrEd* by Simonetto, Archambault, Auber, and Bourqui [26]. There is also another redrawing algorithm that was introduced by Brown and Baird [28]. Even though their algorithm is specifically designed for the Game of Sprouts, we decided to base our algorithm on ImPrEd, since it is described in more detail and force-directed algorithms are simple and easy to implement.

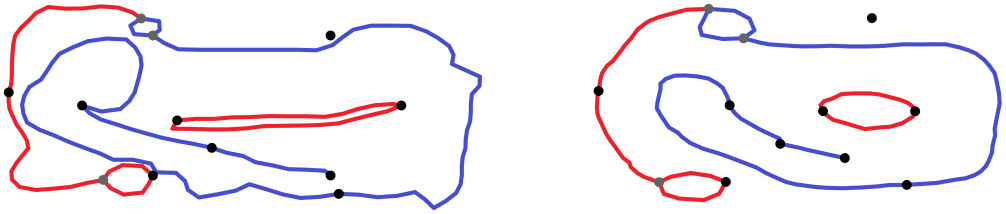
ImPrEd is an iterative force-directed algorithm that improves the existing graph drawing G while preserving its edge-crossing properties. At each iteration, it computes a force $F(v) \in \mathbb{R}^2$ for each vertex v in the graph G that depends on the position of other vertices and edges of G. At the end of the iteration, each node v is moved in the direction $F(v)$. However, the amplitude of each move is restricted so that the edge-crossing properties are preserved.

We introduce our own variant of the algorithm ImPrEd that we use to improve a drawing of a position P. We let V be the vertex set of P, V_G be the set of game vertices of P, E be the set of edges of P, and E_s be the set of small edges of P. Among the main differences with respect to ImPrEd, we modify the computation of the forces as described in Subsection 2.2.1. To compute the zones of possible locations of the vertices, we use the same approach as Bertauls [27], which is briefly described in Subsection 2.2.2. To obtain even better drawings, we also implement merging and subdividing of the edges, similarly as proposed by Brown and Baird [28]. This is described in Subsection 2.2.3 together with some additional features that improve the performance of our algorithm. We summarize all steps of the algorithm in Subsection 2.2.4.

The effect of our redrawing algorithm is illustrated in Figure 2.8. There is a Sprouts position before redrawing and the resulting drawing after 30 iterations of the redrawing algorithm. The position consists of roughly 150 small edges and it took only hundreds of milliseconds to compute. So the algorithm is suitable for real-time computations.

2.2.1 Types of forces

Both the algorithms PrEd and ImPrEd use three different kinds of forces between vertices and edges: the attraction force F^a between vertices that are connected by a small edge, the repulsion force F^r between each pair of vertices, and the repulsion force F^e between vertices and edges. The basic idea of this choice of forces is that the repulsion forces F^e and F^r guarantee enough space between two



Before redrawing. After redrawing.

Figure 2.8: Redrawing a Sprouts position with 30 iterations.

edges and vertices, respectively. The attraction force F^a adjusts the length of the edges to a default value.

The first two features offer exactly what we expect from our redrawing algorithm. However, the last feature does not meet with our criterion that we want to shorten too long edges consisting of many smaller ones. The problem is that our edges are subdivided into small edges with a default length $\delta > 0$. Thus, if we counted the repulsion forces between any pair of vertices, all the small edges would be roughly of the length δ and that would keep the whole edge of roughly the same length, which might be too long or too short. Therefore, we do not count the repulsion forces between vertices on the same edge. However, to regulate the length of an edge e somehow, we count the repulsion forces between game vertices that lie on e. To spread out vertices, we count these forces also between any two vertices of distinct edges. Finally, in our algorithm, there are four additional static edges around the game board, which prevent vertices to move outside of the playing area $[0, 1]^2$.

In the definitions of forces, we use three different constants $\beta_{u,v}$, γ, and δ that customize the balance of these forces. The constant $\gamma > 0$ is the optimal distance of edges and δ is equal to the optimal length $l_{opt} > 0$ of a small edge. The value of $\beta_{u,v} > 0$, which depends on two vertices u and v, is mainly used to strengthen the repulsion force between u and v if u and v are adjacent game vertices. The precise value of the attraction force $F^a \colon \binom{V}{2} \to \mathbb{R}^2$ is given by the following formula

$$F^a(u, v) = \frac{\|u - v\|}{\delta}(u - v).$$

The repulsion forces $F^r \colon \binom{V}{2} \to \mathbb{R}^2$ and $F^e \colon V \times E_s \to \mathbb{R}^2$ are given by

$$F^r(u, v) = \begin{cases} \left(\frac{\beta_{u,v}}{\|u-v\|}\right)^2 (v - u) & \|u - v\| < \beta_{u,v} \\ (0, 0) & \text{otherwise,} \end{cases}$$

and

$$F^e(v, ab) = \begin{cases} \frac{(\gamma - \|v - v_{ab}\|)^2}{\|v - v_{ab}\|}(v - v_{ab}) & v_{ab} \in ab, \|v - v_{ab}\| < \gamma \\ (0, 0) & \text{otherwise,} \end{cases}$$

where v is a vertex disjoint from the edge ab and the point v_{ab} is the projection of the vertex v onto the line defined by the small edge ab.

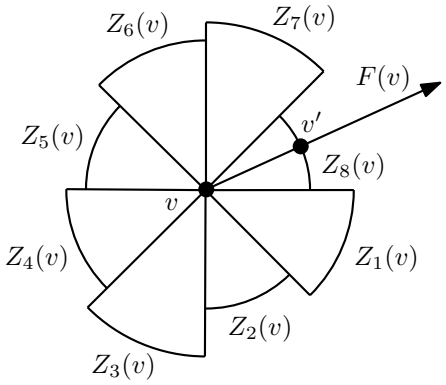


Figure 2.9: A zone $Z(v)$ together with a new location v' of v moved in the direction $F(v)$.

Let S be the set of the four fixed edges representing the playing area. Then, the overall force $F\colon V \to \mathbb{R}^2$ is computed for each vertex v as follows[1]

$$F(v) = \sum_{\substack{u \in V \\ uv \in E_s}} F^a(u,v) + \sum_{\substack{u \in V \setminus \{v\} \\ \nexists e \in E\colon uv \in e}} F^r(u,v) + \sum_{ab \in E_s \cup S} F^e(v,ab) - \sum_{\substack{u,w \in V \\ vw \in E_s}} F^e(u,vw).$$

2.2.2 Moving a vertex

After each force is computed, the vertices are moved in the direction of the resulting force $F(v)$. Since the redrawing algorithm has to preserve edge-crossings, we need to restrict the amplitude of these movements.

In our program, we proceed in the same way as in PrEd [27]. For each vertex v, we define a *zone* $Z(v)$ as a union of eight circular sectors $Z_1(v), \ldots, Z_8(v)$ with radii $r_1(v), \ldots, r_8(v)$, respectively; see Figure 2.9. The zone $Z(v)$ is used to restrict the movement of v in the direction $F(v)$. At the end of each iteration of the redrawing algorithm, we compute the zone of each vertex v, find a sector $Z_i(v)$ that intersects the computed force $F(v)$, and then we move v in the direction $F(v)$ by $\min\{\|F(v)\|, r_i(v)\}$.

We need to set the radii $r_1(v), \ldots, r_8(v)$ so that the edge-crossings are preserved. To compute the radii of a zone $Z(v)$, we only consider small edges ab of P that are disjoint from v. Based on each such a small edge ab, we restrict the appropriate radii of the three vertices v, a, and b. The initial values of the radii are set to a maximum size M_{max} of a movement within a single iteration. The next step of the computation depends on the position of the projection v_{ab} of the vertex v onto the line determined by the edge ab. We distinguish the following two cases:

1. $v_{ab} \in ab$: Let $d = \|v - v_{ab}\|$ and let $Z_s(v)$ be the sector that intersects the ray starting at v and going through v_{ab}. Then, we update the zones of v, a, and b by letting

$$\begin{aligned} r_j(v) &= \min\{r_j(v), d/3\}, & j &= f(s-2), \ldots, f(s+2) \\ r_j(a) &= \min\{r_j(a), d/3\}, & j &= f(s+2), \ldots, f(s+6) \\ r_j(b) &= \min\{r_j(b), d/3\}, & j &= f(s+2), \ldots, f(s+6), \end{aligned}$$

[1] We note that our formula of $F(v)$ has opposite signs at F^a and F^r when compared to the corresponding formula used by Bertauls [27] and by Simonetto et al. [26]. However, we believe that their choice is incorrect.

where $f(x) = ((x-1) \mod 8) + 1$.

2. $v_{ab} \notin ab$: Let $d_a = \|v - a\|$ and $d_b = \|v - b\|$. Then, we update the zones by letting

$$\begin{aligned} r_j(v) &= \min\{r_j(v), d_a/3, d_b/3\}, && j = 1, \ldots, 8 \\ r_j(a) &= \min\{r_j(a), d_a/3\}, && j = 1, \ldots, 8 \\ r_j(b) &= \min\{r_j(b), d_b/3\}, && j = 1, \ldots, 8. \end{aligned}$$

The proof of the correctness of the PrEd algorithm by Bertauls [27] depends only on the zones and not on a particular choice of the forces. Since we defined the zones in the same way as in the PrEd algorithm, the proof of the correctness of our algorithm is the same and we do not state it here.

2.2.3 Improvements

In this subsection, we introduce additional features that improve the resulting drawings of our redrawing algorithm and also its performance.

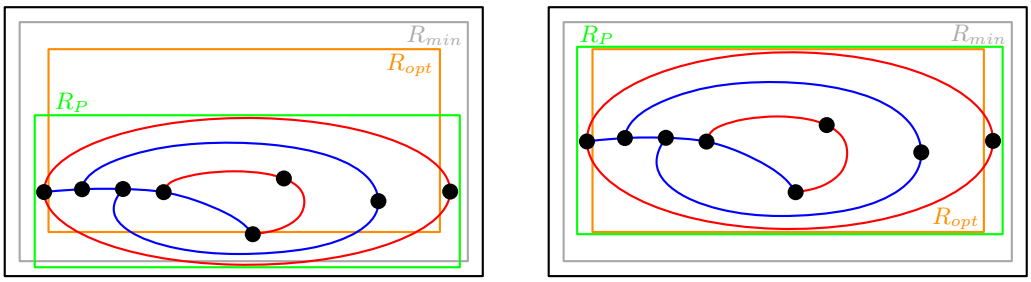
Merging and subdividing small edges Since we do not compute the repulsion forces between inner vertices on the same edge, the length of small edges can become much smaller than the optimal length l_{opt}. However, the attraction force becomes weaker with the decreasing length of the small edge, which can thus produce edges subdivided into too many small edges and still not being short enough. To eliminate this behavior, we merge too short small edges together after each iteration. If the merging algorithm finds a pair of two adjacent small edges uv and vw such that the distance $\|u - w\|$ is below a certain value l_{mer}, then the small edges uv and vw are replaced by a new small edge uw. To prevent crossings, we merge uv and vw only if there is no vertex of P in the triangle uvw. After uv and vw are merged, the length of uw is at least the sum of the lengths of uv and vw, which allows us to shorten uw more easily than uv and vw separately.

There is also a problem when a drawn edge e is too short and it should be elongated. After few iterations of the redrawing algorithm, the length of the small edges of the edge e can be much greater than the optimal length l_{opt} which makes further elongation of e more difficult. In this case, we subdivide the small edges of e whose length is greater than a certain value l_{sub}. This makes subsequent elongation of the subdivided edges easier.

Stretching a drawing Now, we describe an algorithm that improves drawings in two aspects. The first one is avoiding edges being too close to the border of the playing area. There are already repulsion forces between the border and vertices of the drawn position that could solve the problem. However, these forces cannot ensure that there will never be such edges because the repulsion force between them and the border can be suppressed by other forces. Second, we want to keep the drawings suitably stretched out to cover as much of the playing area as possible.

The algorithm that fulfils these requirements is relatively simple. Consider a position P and the rectangle R_P that is circumscribed to the position P. To define

the margin m_{min} of the playing area in which any edge cannot occur we define the rectangle $R_{min} = [m_{min}, 1 - m_{min}]^2$. For $m_{opt} > m_{min}$, we define another rectangle $R_{opt} = [m_{opt}, 1 - m_{opt}]^2$ to which the position should be stretched out; see Figure 2.10. First of all, the algorithm adjusts the coordinates of vertices of P so that the rectangle R_P is centralized. The second step to recompute the coordinates so that R_P is contained in R_{min}. This satisfies the first requirement. To solve the second requirement, we stretch out R_P by adjusting the coordinates of vertices of P to the size of R_{opt} but only in the direction in which R_P does not exceed R_{opt}. To avoid too fast expansions of P, we do not stretch out R_P fully to R_{opt} but only by a certain value s_{max} in each iteration of the redrawing algorithm.



Before stretching P. After stretching P.

Figure 2.10: A position P that was redrawn by the stretching algorithm.

Performance improvements So far, we have not discussed the computational complexity and the performance of the redrawing algorithm. The algorithm consists of three parts: the computation of the forces, the computation of the zones together with moving of the vertices, and the two adjustments of the drawing from the previous paragraphs. The computational complexity of these parts is $\mathcal{O}(|E_s|+|V|^2+|V||E_s|)$, $\mathcal{O}(|V||E_s|)$, and $\mathcal{O}(|E_s|+|V|)$, respectively. Since $E_s = \mathcal{O}(|V|)$, the overall computational complexity of the redrawing algorithm is $\mathcal{O}(|V|^2)$.

The performance of the redrawing algorithm without any improvements is on the verge of usability in real-time computations. We would be able to run only a few number of iterations on a position with a large number of vertices. Thus, we need to improve the performance of the redrawing algorithm. The first idea is to update zones of vertices only for small edges that lie in the same region. This approach highly improves the performance of the computation of zones in positions with high number of regions. However, it does not have any effect on positions with a single region.

Therefore, we use a popular graphical data structure *quadtree* to speedup the computation as proposed by the authors of ImPrEd [26]. A quadtree is a tree data structure whose inner nodes have exactly four children. It is used to partition a plane area by recursively subdividing it into four quadrants. We use this structure to store positions of vertices and small edges. The benefit of using quadtrees is that it allows us to choose stored elements that are close to a certain point in sublinear complexity.

We store vertices into leaves of a quadtree T so that each leaf contains at most one vertex. The initial state of T consists of only a single quadrant that represents

the whole playing area. Then, to insert a vertex v, we look for a leaf quadrant Q that contains v. If Q is empty, we insert v into it. Otherwise, Q is subdivided into four new *subquadrants* that become leaf nodes and children of Q. Then, v is inserted to the new created quadrant that contains v. For a better understanding, see Figure 2.11 where the hierarchical structure of a quadtree that contains nine vertices is shown.

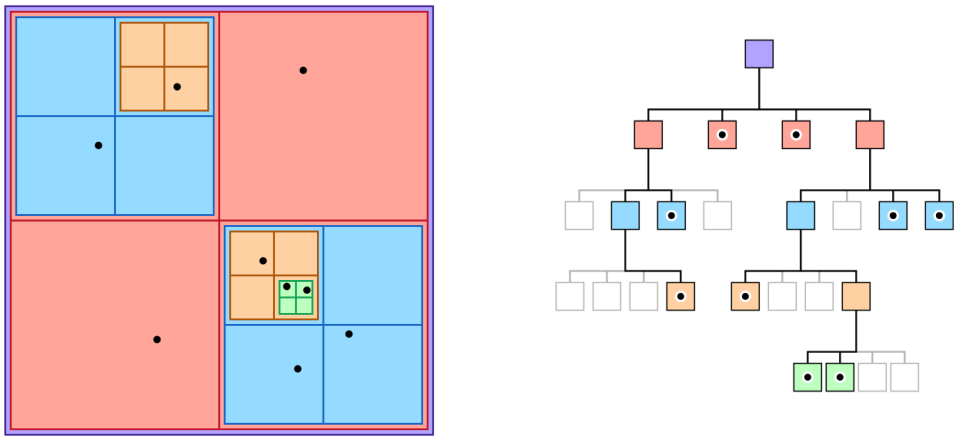


Figure 2.11: A quadtree that contains nine vertices (taken from [29]).

A quadtree T that stores vertices allows us to efficiently find vertices that are at distance at most d from a point p. Equivalently, we look for vertices $v \in T$ that are inside the circle with the center in p and the radius d. In our program, we approximate this task by looking for vertices that are inside the square S with the center p and with the sides of length $2d$. Searching for such vertices in S is described in Algorithm 2. It follows from the algorithm that if a quadrant Q does not intersect S, we do not have to explore Q any further. This is the main advantage of quadtrees that leads to a sublinear complexity of searching for vertices and also of vertex insertion.

In our program, we also apply quatrees to small edges. The problem here is that the small edges do not have to fit into a single leaf quadrant but they can lie across multiple quadrants. To solve this, we allow to store a small edge s in an inner quadrant Q if s does not fit into any subquadrant of Q. Then, the algorithm that finds small edges close to a certain point p is almost the same as Algorithm 2. The only two differences are that a small edge s is not only close to the point p if it is contained in the square S but also if it intersects S. The second difference is that since small edges can be inside an inner quadrant, we need to explicitly check the mutual position of S and the small edges that are inside the currently explored inner quadrant Q. This solution slightly reduces the advantage of using quadtrees, however, this negative effect is not significant since our small edges are relatively small.

As we are already able to insert vertices and small edges to quadtrees and effectively find vertices and small edges that are close to a certain point, we can improve the performance of our redrawing algorithm as follows. Since the amplitude of the movements of vertices is bounded by the finite value M_{max}, we can safely update the zones of a vertex v using only small edges s whose distance

Algorithm 2: Find all vertices inside a quadrant Q that are inside a square S

```
Function FindVertices(Q, S)
    if Q is a leaf node then
        if Q contains a vertex v ∈ S then
            return {v};
    else
        V ← {};
        foreach subquadrant Q_s of Q do
            if Q_s is contained in S then
                U ← the set of vertices in Q_s ;
                V ← V ∪ U;
            else if Q_s intersects S then
                U ← FindVertices(Q_s, S) ;
                V ← V ∪ U;
        return V;
```

from v is less than $2M_{max}$. Furthermore, we only need to consider pairs of v and s that are in the same region. Thus, we construct a quadtree for each region separately and we later use them for updating zones within the corresponding regions.

The second application of quadtrees is during the computation of forces. We compute forces only between elements that are close enough. More precisely, we compute the repulsion force $F^r(u, v)$ between vertices u and v only if their distance is less than $\beta_{u,v}$. For the edge repulsion force $F^e(v, ab)$ between a vertex v and a small edge ab, we similarly compute the force only if the distance of v and ab is less than γ. To determine close pairs of two vertices and close pairs of a vertex and a small edge, we use a global quadtree of vertices and a global quadtree of edges. Recall that we approximate distances by the squares circumscribed to the corresponding circles since this simplification has a minimal impact on the final drawing.

Finally, to ensure that the redrawing procedure does not take too long, we set a time limit after which the redrawing is terminated even if not all iterations have finished.

2.2.4 Summary

Now, as we have already fully described our redrawing algorithm, we summarize the exact sequence of all the steps of the redrawing algorithm in Algorithm 3.

In the previous subsections, we introduced many constants that adjust the behaviour of the redrawing algorithm without mentioning their real values. Since the correctly balanced values of the constants are the core of the redrawing algorithm, we list all the constants with the values that are used in our program in Table 2.2.

Note that the width and the optimal length of small edges depend on the

Algorithm 3: Redrawing of a position P using n iterations

```
Function Redraw(P, n)
    foreach i = 1, ..., n do
        construct quadtrees;
        foreach v ∈ V do
            compute the force F(v);
            compute the zones Z(v);
        move the vertices;
        merge the small edges;
        subdivide the small edges;
        stretch the position P;
```

number of game vertices in a position. This is because a position runs out of available space with the increasing number of game vertices, or equivalently with the increasing number of moves. Thus, we need to gradually decrease the width of small edges which saves some space. We also need to shorten the length of edges since we have to make drawings finer as the position contains a higher number of vertices.

Note that $\beta_{u,v}$ can acquire four different values depending on vertices u and v. In general, the repulsion force is amplified if one of the vertices is a game vertex. To make the edges long enough, the repulsion force is strengthened even more between game vertices u and v on the same edge. Furthermore, to make edges in the outer region longer than in inner regions, we make these forces the strongest in this case.

2.3 Computer opponent

To reach Goal **G3**, we need to implement a strong Sprouts solver to create a perfectly playing computer opponent. Thus, we include algorithms that are inspired by the currently best Sprouts solver GLOP [6] and that are discussed in the following subsections. In Subsection 2.3.1, we describe a data structure that captures game properties of Sprouts positions in a very simple manner. Then, in Subsection 2.3.2, we discuss mechanisms how to reduce the number of possible representations of a single position by using a suitable encoding of the positions. We describe the method for generating children of a position in Subsection 2.3.3. In Subsection 2.3.4, we apply results from the theory of nimbers to efficiently traverse a game tree of a Sprouts position in order to determine the outcome and to find a best move. In our program, we implemented parallelism, described in Subsection 2.3.5, to traverse the game tree of a position. Finally, in Subsection 2.3.6, we show how to generate a database of pre-trained positions for perfectly playing opponents that will never take too long to find a best move.

Symbol	Name	Value
w	the width of a small edge	$0.023 - 0.0004 \cdot \lvert V_G \rvert$ $\lvert V_G \rvert \leq 40$ 0.007 otherwise
l_{opt}	the optimal length of a small edge	$0.045 - 0.0008 \cdot \lvert V_G \rvert$ $\lvert V_G \rvert \leq 40$ 0.013 otherwise
l_{mer}	the limit length for merging adjacent small edges together	l_{opt}
l_{sub}	the limit length for subdividing a small edge	$1.8 \cdot l_{opt}$
δ	the constant for the attraction force	l_{opt}
$\beta_{u,v}$	the constant for the repulsion force between vertices u and v	4δ $uv \in E$ and $uv \subseteq R_O$ 3δ $uv \in E$ and $uv \not\subseteq R_O$ 2δ $uv \notin E$ and $\{u, v\} \cap V_G \neq \emptyset$ δ otherwise
γ	the constant for the edge-vertex repulsion force	$10 \cdot w$
M_{max}	the maximal movement in a single iteration	0.01
m_{min}	the minimal stretching margin	0.04
m_{opt}	the optimal stretching margin	0.1
s_{max}	the maximum size of stretching out	0.02

Table 2.2: The values of the drawing and redrawing constants.

2.3.1 String representation

In Section 2.1, we described the graphical representation that is used to work with a drawing of a Sprouts position. However, since the graphical representation contains a lot of graphical-specific information, such as coordinates of vertices, small edges that form the whole edges, and the hierarchical structure of regions, it is not suitable for the analysis of Sprouts positions whose representation has to be as simple as possible to express its game properties.

Therefore, we introduce another representation of a position P, called the *string representation*, which is denoted by $sr(P)$. This structure is used in many other papers analyzing Sprouts [5, 6, 12] and it is actually a simplification of the graphical representation. We describe the individual parts of the string representation in the following paragraphs.

We will not explicitly distinguish between vertices, boundaries, regions and their string or graphical representations unless it is necessary.

Vertices In the string representation, we store only game vertices. We do not store inner vertices, in fact we even do not store edges, since it is not important to keep the exact shape of a position for determining its outcome. The string representation $sr(v)$ of a vertex v is denoted by a capital letter.

Boundaries A boundary β of P is represented by the string $sr(\beta) = V_1 \cdots V_n$ of the game vertices of β in the order we meet them as we traverse β. In the string representation, we do not distinguish between inner and border boundaries, thus $sr(\beta)$ can have either the clockwise or the counterclockwise orientation. Note that $sr(\beta)$ corresponds to a certain rotation or a reversion of the vertex-sequence of $gr(\beta)$. Each letter V_i is called a *vertex occurrence* in β.

Regions A region R is represented by the string $sr(R) = s(\beta_1).\cdots.s(\beta_n).$ of the strings $sr(\beta)$ where β is a boundary of R. Note that the strings of boundaries are terminated by dots. From the same reason as in the graphical representation, the string representations of inner boundaries within the same region must have the same orientation while the string representation of the border boundary must have the opposite orientation. Unlike in the graphical representation, the regions are not organized into any hierarchical structure in the string representation.

Lands A new structure, which is very important in the analysis of Sprouts positions, are *lands*. Informally, lands are mutually independent parts of a position. A land L is represented by the string $sr(L) = sr(R_1)|\cdots|sr(R_n)|$ where $sr(R_i)$ are strings of regions of L.

To describe lands of a position P formally, we define a binary relation $\sim$ on the set $\mathcal{R}$ of regions of P. For regions $R, S \in \mathcal{R}$, we let $R \sim S$ if $R = S$ or if there is an alive game vertex incident to both R and S. We let $\equiv$ be the equivalence relation on $\mathcal{R}$ obtained as the transitive closure of $\sim$. The lands of P are then the equivalence classes of $\equiv$ on $\mathcal{R}$.

We describe a method how to construct all lands of a position P in Algorithm 4. We proceed by storing the current set of lands which is gradually updated. Then, we go through regions of P which results in a new land consisting of the current

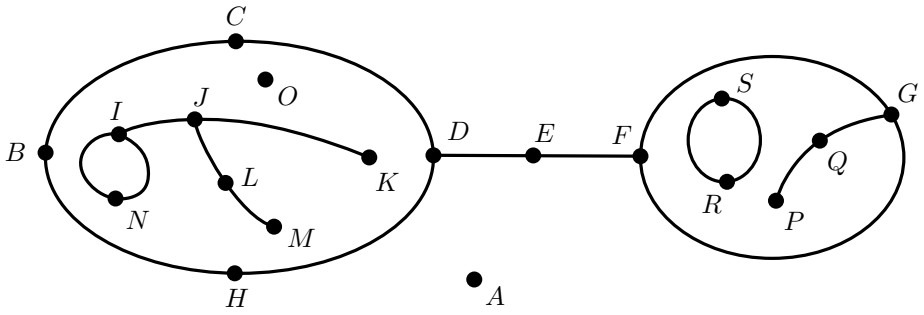


Figure 2.12: A position with game vertices labelled A–S.

region R and regions of lands that share an alive vertex with R. We update the set of current lands by replacing all the lands sharing an alive vertex with R by the new merged land.

Algorithm 4: Constructing strings of all lands of P

```
Function ConstructLands(P)
    R ← the set of strings of regions of P;
    L ← {};
    foreach R ∈ R do
        L' ← {};
        MergedLand ← R;
        foreach L ∈ L do
            if L shares alive game vertex with R then
                MergedLand ← MergedLand | L;
            else
                L' ← L' ∪ {L};
        L' ← L' ∪ {MergedLand|};
        L ← L';
    return L;
```

Position A position P is represented by the string $sr(P) = s(L_1)+\cdots+s(L_n)+!$ of the strings $sr(L_i)$ of the lands L_i of P.

In Figure 2.12, we can see the position P that has already been used in Subsection 2.1.1 for describing the graphical representation. The string representation of the position P can be represented by the following string:

```
A.BCDEFGFEDH.|DCBH.IJKJLMLJIN.O.|IN|+PQGFGQ.RS.|RS.|+!
```

Note that each terminating character (`.`, `|`, `+`, or `!`) that is not between two structures on the same level can be removed. However, GLOP does not remove them during the computation. We also do not remove these characters to keep the results consistent with the outputs of GLOP. Nevertheless, for better readability, we remove these terminating characters in the text and in the databases.

Moves As in the graphical representation, there are the same two types of moves that can be inserted into the data structure. The inserting algorithms are much simpler than in the graphical representation since we do not have to care about the orientations of the inserted edges nor about updating the hierarchical structure of regions. In the following algorithms, we assume that the vertices we are connecting have enough lives.

Double-boundary moves Consider two different boundaries with sequences $A_1 \cdots A_i \cdots A_m$ and $B_1 \cdots B_j \cdots B_n$. Then, the double-boundary move that connects two occurrences A_i and B_j creates a new boundary $A_1 \cdots A_i Z B_j \cdots B_n$ $B_1 \cdots B_j Z A_i \cdots A_m$ where Z is the newly added vertex.

Single-boundary moves Let β be a boundary with the sequence $A_1 \cdots A_i \cdots A_j \cdots A_n$ in a region R with boundaries from R partitioned into the *major partition* $\mathcal{B}_{major}$, the *minor partition* $\mathcal{B}_{minor}$, and $\{\beta\}$. Consider the single-boundary move that connects two (not necessarily different) occurrences A_i and A_j and that separates the boundaries $\mathcal{B}_{major}$ from the boundaries of $\mathcal{B}_{minor}$. Then, the move splits the original region R into the *major region* with the string $A_i \cdots A_j Z.\mathcal{B}_{major}$ and into the *minor region* with the string $A_1 \cdots A_i Z A_j \cdots A_n.\mathcal{B}_{minor}$. The newly created boundaries are called the *major boundary* and the *minor boundary*, respectively. Note that we need to distinguish between $\mathcal{B}_{major}$ and $\mathcal{B}_{minor}$ since swapping them in the major and minor regions can create a non-equivalent string.

2.3.2 Simplifying strings

In the string representation, a single Sprouts position can be represented by many different strings which leads to repetitive computations during the exploration of the game trees. For example, the position P from Figure 2.12 can be represented by the already mentioned string

```
A.BCDEFGFEDH|DCBH.IJKJLMLJIN.O|IN+PQGFGQ.RS|RS
```

or it can also be expressed by the following string that differs from the previous one only in the order of the lands:

```
PQGFGQ.RS|RS+A.BCDEFGFEDH|DCBH.IJKJLMLJIN.O|IN
```

The representations of P can differ in many other factors, like labels of vertices or the order of boundaries, regions and so on.

We say that two positions are *equivalent* if they are isomorphic as plane embeddings. Note if two positions are equivalent then they have the same game tree. Also note that two equivalent positions can be represented by many different strings.

Therefore, we introduce methods that simplify the string representation and that reduce the number of possible strings representing equivalent positions. We use two methods called the *string reduction* and the *string canonization*. The string reduction serves for deleting unnecessary parts of a string and for simplifying names of some vertices by assigning them special values. While the string reduction mainly simplifies a string s, the string canonization tries to find a representative string of the equivalence class of s to eliminate different but equivalent strings. This done by reordering s and by relabeling its vertices.

String reduction The string reduction simplifies strings using the following five steps. The input of the reduction is a string of a position which might not be fully partitioned into lands. The correct partition into lands is one of the steps of the reduction method. We describe the method using the following string s that represents the position P from Figure 2.12 and that is not partitioned into lands yet.

`A.BCDEFGFEDH|DCBH.IJKJLMLJIN.O|IN|PQGFGQ.RS|RS`

1. **Delete dead parts**: First, we delete all dead vertices from the string. Those are the vertices whose letter occurs three times within a land of the current land-partitioning of s. Then, we delete boundaries that become empty after deleting their dead vertices. Then, we delete dead regions (recall that those are the regions R with $l(R) \leq 1$). Finally, we delete all empty lands. The string s and the position P after the deletion are in Figure 2.13.

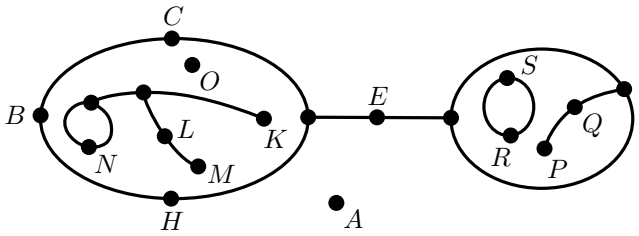


Figure 2.13: `A.BCEEH|CBH.KLMLN.O|PQQ.RS|RS`

2. **Apply generic names**: In this step, we assign *generic names* to vertices whose occurrences do not have to be distinguished by capital letters. First, we rename each singleton to `0`. Then, we rename each vertex with two lives to `1` since it occurs in the string only once.

   Before applying the third type of generic names, we merge vertices that occur twice in a row to a single occurrence as such doubled occurrences do not hold any extra information. See the doubled occurrence of `E` and `Q` in our example. Then, we rename each letter that occurs only once in the position to `2`. These are the vertices with a single life that used to occur twice in the position but their second occurrence was deleted by the previous step or during deleting dead regions. The string s and the position P after applying the generic names are in Figure 2.14.

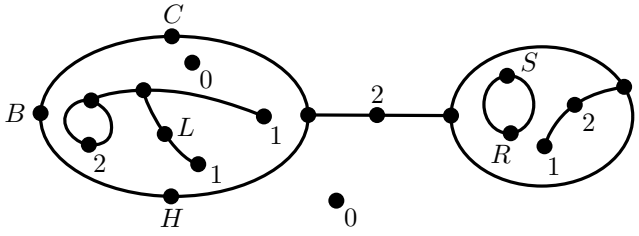


Figure 2.14: `0.BC2H|CBH.1L1L2.0|12.RS|RS`

3. **Split lands**: Here, we use Algorithm 4 to split the position into its independent lands. The string s is then `0.BC2H|CBH.1L1L2.0+12.RS|RS`.

4. **Rename letters**: So far, we have only used a sequence of capital letters globally in the whole string. However, we can use the same letters in two lands independently since lands do not share any vertices. Moreover, we can use lowercase letters locally for vertices that occur only in a single boundary, which reduces the number of capital letters that need to be used.

   In this step, we rename vertices contained only in a single boundary starting from `a` within this boundary in the order they occur in the string. We rename all other letter vertices within a single land starting from `A` in the order they occur in the string. The string s and the position P after the renaming of the letters is in Figure 2.15.

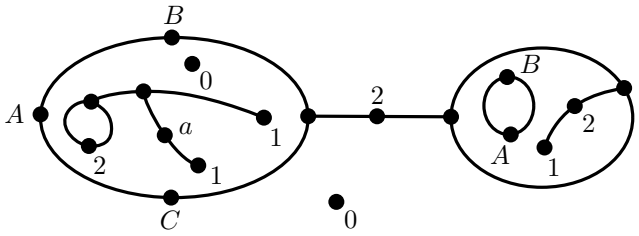


Figure 2.15: `0.AB2C|BAC.1a1a2.0+12.AB|AB`

5. **Merge boundaries**: The last step of the reduction is used only for regions R with $l(R) \leq 3$. Each position P with such a region R has the same game tree as the position that is obtained from P by merging the boundaries of R, which can be proved using a small case analysis. Thus, we merge boundaries of such regions. This step considerably reduces the number of positions that need to be explored during the analysis of a position. However, the trade-off of this improvement is that it significantly complicates the synchronization of the string and the graphical representation in our program as we will see in Section 2.4.

String canonization Even after a string that represents a position is reduced, there can be many other reduced strings that represent equivalent positions. To identify such strings, we define an equivalence $\equiv$ on the set S of strings of P. For strings $s_1, s_2 \in S$, we let $s_1 \equiv s_2$ if one string can be obtained from the other by a sequence of the following steps:

a) rotating the sequence of the vertices in a boundary,

b) reversing the orientation of the boundaries in a region,

c) reordering the boundaries in a region,

d) reordering the regions in a land,

e) reordering the lands in a position,

f) renaming lowercase and capital letters.

Consider strings s_1 and s_2 that represent positions P_1 and P_2, respectively. Then, we have $s_1 \equiv s_2$ if P_1 and P_2 are equivalent.

The goal of the *full string canonization* of a string s is to replace s by a unique representative string from its equivalence class $[s]_{\equiv}$. If we determine such a representative string for each equivalence class then we eliminate repetitive computations over equivalent positions. The full string canonization chooses as the representative string the minimal string in the lexicographical order of S induced by the following ordering of the characters:

$$\texttt{0} < \texttt{1} < \texttt{2} < \texttt{a} < \texttt{b} < \cdots < \texttt{A} < \texttt{B} < \cdots < \texttt{.} < \texttt{|} < \texttt{+} < \texttt{!}$$

The fully canonized string from our example is the following one:

```
0.12a1a.ABC|0.2ABC+12.AB|AB
```

Unfortunately, we cannot afford the full canonization of strings since it requires too much running time. This is because we have to consider all the possible labelings of vertices in the full canonization. For instance, if a land in a position contains k different capital letters, then the full canonization goes through at least $k!$ strings. According to Lemoine and Viennot [6], the full canonization is too demanding to compute outcomes of the n-spot positions with $n \geq 5$.

Therefore, we only perform so-called *pseudo-canonization* that does not necessarily find the optimal representative string. The pseudo-canonization of a string s modifies the string s using only Steps a)–e) without renaming its vertices. In the pseudo-canonization, the vertices of s are first renamed as in Step 4 of the string reduction and then we sort s. This step is performed twice in the pseudo-canonization. This is the most efficient approach according to Lemoine and Viennot [6].

The example of equivalent positions in Table 2.3 shows that there exist equivalent strings that are not distinguished by the pseudo-canonization. Lemoine and Viennot [6] claim that this problem does not occur often in real computations of outcomes since it mainly explores positions with a low number of capital letters.

However, this does not necessarily hold for our application. To implement a perfectly playing computer opponent, we often need to explore segments of the game tree that do not have to be explored in the computations of an outcome. Therefore, we can meet strings with a high number of capital letters more frequently. This results in relatively large databases with many equivalent strings produced by our program. Nevertheless, the sizes of the databases are still feasible for implementing the computer opponent.

ABCDEF|ABCDEG|FG
ABCDEF|ABCDGF|EG
ABCDEF|ABCGEF|DG
ABCDEF|ABGDEF|CG
ABCDEF|BCDEFG|AG

Table 2.3: An example of equivalent strings that are not distinguished by the pseudo-canonization during the exploration of the game tree of the 4-spot position (taken from [6]).

Note that all the singletons are always located at the beginning of the string of a region after the canonization or the pseudo-canonization are performed. Therefore, we introduce the following notation that shortens the string $0.\cdots.0.\beta_1.\cdots.\beta_m$ with n singletons in a row by the string $0^*n.\beta_1.\cdots.\beta_m$. This notation is only used in the text and in the databases and not in the string representation nor in its simplification.

Table 2.4 contains the numbers of distinct strings in the game trees of the n-spot positions with $n \leq 6$. The numbers in our table match the numbers produced by GLOP since our solver is almost the same except of some additional features that are necessary for the synchronization with the graphical representation. For comparison, the 10-spot position would have 5165 different opening moves without any string reduction or canonization while only 5 of them are mutually non-equivalent.

n	number of strings
2	18
3	157
4	1796
5	24784
6	393103

Table 2.4: The sizes of the game trees of the n-spot positions for $n \leq 6$.

2.3.3 Generating children of a position

Before we introduce the main algorithm that computes the outcome of a position, we describe an algorithm that generates children of a position. The algorithm should ideally produce only pairwise non-equivalent children since the simplification of the extra generated children consumes time and it can cause repetitive computations.

Generating only pairwise non-equivalent children is very idealistic. Browne [5] suggested a method called *boundary matching* which reduces the number of generated children by not distinguishing between boundaries with the same string. It turned out that this method improves the running time the most when we do not distinguish only singletons as they are easy to detect and they appear quite often. This technique was also used in GLOP [6] and we apply it as well.

Let P be a *simplified* position. That is, a position to whose string we already applied the string reduction and the string pseudo-canonization. The process of generating children of P is shown in Algorithm 5. We distinguish *double-boundary children* and *single-boundary children* according to the type of the move that creates them. Generating the double-boundary and the single-boundary children of P is described in the functions $GenerateDBChildren(P)$ and $GenerateSBChildren(P)$, respectively. The algorithm does not distinguish singletons as P is simplified, and thus all singletons occur at the beginning of a region R.

Algorithm 5: Generate children of a simplified position P

```
Function GenerateDBChildren(P)
    foreach land L of P do
        foreach region R of L do
            B ← the list of boundaries of R;
            startingIndex ← 0;
            if B contains at least two singletons then
                startingIndex ← the penultimate singleton index in B;
            for i ← startingIndex to length(B) do
                for j ← i to length(B) do
                    α ← B[i];
                    β ← B[j];
                    foreach occurrence A_k in α do
                        foreach occurrence B_l in β do
                            generate the double-boundary child of P by
                            the double-boundary move connecting A_k and B_l;
Function GenerateSBChildren(P)
    foreach land L of P do
        foreach region R of L do
            B ← the list of boundaries of R;
            startingIndex ← 0;
            if B contains at least two singletons then
                startingIndex ← the last singleton index in B;
            for i ← startingIndex to length(B) do
                β ← B[i];
                P ← the set of all possible partitions of B \ {β} without
                 distinguishing singletons;
                for j ← 0 to length(β) do
                    for k ← j to length(β) do
                        A_j ← the jth occurrence in β;
                        A_k ← the kth occurrence in β;
                        if j ≠ k ∨ l(A_j) ≥ 2 then
                            foreach (B_major, B_minor) ∈ P do
                                generate the single-boundary child of P by
                                 the single-boundary move connecting A_j and
                                 A_k and partitioning B into B_major and
                                 B_minor;
```

2.3.4 Computation of outcomes

Now that we are able to represent a position P and to generate its children, we can start exploring the *game tree* of P. By Observation 3, either the first or the

second player has a winning strategy. Therefore, the main goal of the exploration algorithm is to determine the *outcome* of P. If the first player has a winning strategy, then the outcome of P is Win and it is $Loss$ otherwise. Then, P is called *winning* and *losing*, respectively.

A position P is winning if there exists a losing child C of P since then the first player in P can make the move leading to C, and thus forces the second player to start in the losing position C. Analogously, the position P is losing if there is no losing child of P. Then, each move of the first player leads to a winning position in which the second player starts.

By these observations, we can construct a simple Algorithm 6 for determining the outcome of a position P. The algorithm is based on the *Alpha-beta pruning algorithm*. It uses the fact that the outcome of P is Win after we find the first losing child of P independently on the outcomes of the other children that do not need to be explored anymore. In general, determining the outcome of a winning position is much simpler than determining the outcome of a losing position where we need to explore all its children.

Note that the line "$\mathcal{C} \leftarrow$ the children of P" in Algorithm 6 consists of Algorithm 2.3.3 for generating children, the simplification of children described in Subsection 2.3.2, and the filtration of children with equal strings.

Algorithm 6: The Alpha-beta pruning algorithm determining the outcome of a position P

```
Function ComputeOutcome(P)
    C ← the children of P;
    foreach C ∈ C do
        outcome ← ComputeOutcome(C);
        if outcome is Loss then
            return Win;
    return Loss;
```

Algorithm 6 could serve relatively well to implement a perfectly playing computer opponent on positions with a low number of spots. To reach positions with a higher number of spots, we modify Algorithm 6 using improvements introduced by Lemoine and Viennot [6].

Nimber theory Sprouts positions often consist of multiple independent parts that we call lands. Thus, we can try to compute the outcome of each land of a position P separately and then merge these outcomes to determine the overall outcome of P. This can be easily done for pairs of losing lands as then the overall outcome is Loss. Similarly for pairs consisting of a losing and a winning land, the overall outcome is Win. The problem is that the overall outcome of pairs of two winning lands can be either Loss or Win.

Lemoine and Viennot [6] introduced a solution of this issue by computing *nimbers* of lands instead of their outcomes. Using the nimber theory, it is possible to merge these nimbers to determine the outcome of the whole position. This approach allows us to explore lands separately, which reduces the size of the explored game trees in general. To be able to reuse already computed nimbers,

we store them into a *database*. Another advantage of using nimbers is that we only need to store positions with a single land. Without nimbers, we would have to store outcomes of positions that can consist of many lands, which leads to overwhelmed databases [6].

We now state some basics of the nimber theory. *Nim* is an impartial game whose positions consist of heaps, called *Nim-heaps*, of different sizes. In Nim, the players alternate in removing $n \geq 1$ objects from a single heap. The first player who cannot make a move loses the game. In impartial games, this rule is called the *normal play convention*.

Before we describe the relationship between Nim and Sprouts, we define the nimber of a position P of any impartial game.

Definition 1. *The* nimber $|P|$ *of a position P of an impartial game is equal to* $\min \mathbb{N}_0 \setminus \mathcal{N}$ *where $\mathcal{N}$ is the set of the nimbers of the children positions of P.*

Since the nimber of an *empty position*, which is the position with no children, is equal to 0, we can easily prove the following relationship between a nimber and an outcome by induction.

Observation 8. *For a position P of an impartial game, we have $|P| = 0$ if and only if P is losing.* □

Two positions in an impartial game are *equivalent* if no matter what position is added to them, they will have the same outcome. The following result, called the *Sprague–Grundy theorem* [30, 31], shows that each impartial game can be represented as a Nim-heap of certain size $n \in \mathbb{N}_0$.

Theorem 9 (The Sprague–Grundy theorem [30, 31])**.** *Each position P of an impartial game under the normal play convention is equivalent to the one-heap position of Nim with $|P|$ objects.*

By Theorem 9, we can represent a sequence of lands as the sequence of the corresponding Nim-heaps, since lands are independent positions of the impartial game Sprouts. The following result allows us to merge multiple Nim-heaps into a single one, and thus to represent a sequence of lands as a single Nim-heap.

Theorem 10 ([3])**.** *Let h_1 and h_2 be two heaps of the position N of Nim with n_1 and n_2 objects, respectively. Then, the Nim position N' obtained from N by merging h_1 and h_2 into a Nim-heap with $n_1 \oplus n_2$ objects is equivalent to N where $\oplus$ denotes the bitwise exclusive or.*

We do not prove Theorems 9 and 10 here. For a more detailed view of the theory, see the book [3] written by Conway, one of the creators of Sprouts, and others. If we apply the basic results from the nimber theory to Sprouts, we obtain the following result.

Corollary 11. *The nimber of a Sprouts position P consisting of lands $L_1, \ldots, L_n$ is $|P| = |L_1| \oplus \cdots \oplus |L_n|$.*

Couples Now, we can improve Algorithm 6 using the computation of nimbers. First, we generalize Algorithm 6 so that it does not compute the outcome of a position but it computes the outcome of a couple $(P + n)$ where P is a Sprouts position and n is the size of the Nim-heap representing the merged nimbers of lands according to Theorems 9 and 10. A couple $(P + n)$ represents a parallel game of Sprouts on a position P and the game of Nim with a single heap of n objects. Therefore, the set of children of a couple $(P + n)$ is equal to the union of $\{\,(C, n) \mid C \text{ is child of } P\,\}$ and $\{\,(P, m) \mid m < n\,\}$. For example, the outcome of $(\emptyset + n)$ is a *Loss* if $n = 0$ and *Win* if $n \geq 1$. Before describing the algorithm for computing the outcome of a couple, we show how to compute the nimber of a position.

Computing the nimber of a position The nimber of a position P can be computed by applying Definition 1 recursively, that is, by computing the nimbers of all children of P. However, this approach is extremely inefficient. Luckily, there is a simpler way based on the following proposition.

Proposition 12. *The outcome of* $(P + n)$ *is Loss if and only if* $|P| = n$.

Proof. We prove the proposition by the following sequence of equivalences

$$(P + n) \text{ is Loss} \iff |(P + n)| = 0 \iff |P| \oplus n = 0 \iff |P| = n.$$

The first equivalence holds by Observation 8, the second equivalence is implied by Theorems 9 and 10. and the last one follows from the definition of $\oplus$. □

We apply Proposition 12 to efficiently compute the nimber of a position P using Algorithm 7. The algorithm is efficient since it is used only on positions with a single land that usually have low nimbers.

Algorithm 7: Computing the nimber of a position P

```
Function ComputeNimber(P)
    n ← 0;
    while True do
        if ComputeOutcome(P + n) is Loss then
            return n;
        n ← n + 1;
```

Computing the outcome of a couple To determine the outcome of a couple $(P + n)$, we use a similar approach as in Algorithm 6 by computing the outcomes of its children. First, we replace each land whose nimber is known by the corresponding Nim-heap that is merged with the Nim part of $(P + n)$. Then, we compute the unknown nimbers of all remaining lands of P except of one. These nimbers are also merged with the Nim part of the couple. This results in a new couple $(P' + n')$ where the position P' is either the single land whose nimber was not computed or it is the empty position `!`. Finally, we use the updated couple

Algorithm 8: Computing the outcome of a couple $(P + n)$

```
Function ComputeOutcome(P + n)
    merge nimbers of lands whose nimbers are stored in the database with
      the nimber part of (P + n) using the bitwise exclusive or ⊕;
    compute the unknown nimbers of all lands remaining in P except of
      one land P′ and merge them with the nimber part of (P + n);
    (P′ + n′) ← the updated couple (P + n) after the previous steps;
    foreach child C of (P′ + n′) do
        outcome ← ComputeOutcome(C);
        if outcome is Loss then
            return Win;
    store (P′ + n′) into the database;
    return Loss;
```

$(P' + n')$ to compute the outcome of $(P + n)$ since these two couples are equivalent; see Algorithm 8.

By this approach, we always compute the lands separately, and thus we can store only positions with a single land in our database. We do not compute the nimbers of all lands of the position P because it is not necessary and, in general, the computation of a nimber is more complex than the computation of an outcome. Note that the algorithm can be also used to compute the outcome of a standalone Sprouts position P by calling the function with $(P + 0)$.

In Algorithm 8, we only store the outcomes of losing couples, which is equivalent to storing only positions whose nimber is known by Observation 12. We could also store the outcomes of winning couples but it only unnecessarily increases the size of the database since these outcomes can be determined immediately after their children are expanded.

Ordering of children and lands In Algorithm 8, we have an option to choose the order of children in which they will be expanded. For a losing couple, the order of children does not matter, since all its children are winning, and thus need to be explored. However, it is sufficient to find a single losing child for a winning couple. Therefore, there can be multiple different ways how to determine the outcome of the couple and some of these ways can be more computationally demanding that the others.

We estimate the computational complexity of determining the outcome of each child and we order the children by this factor to find a losing child as fast as possible. The computation time of the estimate has to be as small as possible so that we will spend a negligible amount of time during the computations of the outcomes of losing couples as the ordering of their children does not affect the performance.

The order of children is based on the following priorities that are ordered according to the decreasing priority:

- prioritize a couple $(P + n)$ with a smaller value $l(v) + 4n$,
- prioritize a couple $(P + n)$ whose position P has a higher number of lands,

- prioritize a couple $(P + n)$ whose position P has a smaller estimate on the number of children.

In case of a tie for two couples $(P_1 + n_1)$ and $(P_2 + n_2)$, they are ordered lexicographically by their corresponding strings $P_1_n_1$ and $P_2_n_2$.

The last priority only estimates the true number of children of a position P since the computation of the exact value would require too much running time. The algorithm that estimates the number of children is based on the functions shown in Algorithm 9. The functions there estimate the number of double-boundary children between two boundaries β_1 and β_2 and the number of single-boundary children that can be created by a move on a boundary β in a region R. The overall estimate is then computed using these functions with all possible combinations of the parameters. Note that the estimation algorithms distinguish singletons even if they are not distinguished when the children of a position are generated.

Algorithm 9: Estimate the number of children of a position P

Function *EstimateDBChildrenNumber($\{\beta_1, \beta_2\}$)*
- $n_1 \leftarrow$ the number of vertex occurrences in β_1;
- $n_2 \leftarrow$ the number of vertex occurrences in β_2;
- **return** $n_1 \cdot n_2$;

Function *EstimateSBChildrenNumber(β, R)*
- $n \leftarrow$ the number of vertex occurrences in β;
- $\mathcal{B} \leftarrow$ the boundaries of R;
- $p \leftarrow 2^{|\mathcal{B} \setminus \{\beta\}|}$ (the number of partitions of $\mathcal{B} \setminus \{\beta\}$ into two sets);
- **return** $n^2 \cdot p$;

We also choose the order in which the nimbers of the lands of P are computed in Algorithm 8. We apply the priorities used for couples on lands simply by omitting the Nim part of the couple. Then, the land with an unknown nimber that forms the position P' is the land of P with the lowest priority.

These ordering rules are also implemented in GLOP [6] and they significantly improve the performance of the solver according to the authors.

2.3.5 Parallel exploration

Lemoine and Viennot [6] suggested an improvement of the solver by a parallel computation. We have implemented this improvement in our program.

There are multiple ways how the exploration of a game tree of a position P could be parallelized. We chose the approach that parallelizes only the computation of the children of P. The advantage of this approach is that it is simple to distribute the computation between the working threads, and thus fully utilize the available computation power. Consider a position P. If we want to parallelize the computation by using n threads, we gradually assign the children of P to the threads to compute their outcomes. After a thread finishes its job, we assign it the next child in the order. We continue until there is no remaining child or until we find the first losing child, which determines the outcome of P.

The only memory that the threads share with each other is the database. The first simple solution is that each thread would use its own database, and thus there would be no need to synchronize them. This solution works but it has the disadvantage of re-computations since a thread has to compute the nimber of a position that is not stored in its database even if it is already stored in a different one. This trade-off for no synchronization cancels out the advantage of using distributed computations.

Our solution is to use a global database that is equipped by synchronization primitives to prevent deadlocks. Note that the synchronization overhead is not significant here since the synchronized blocks are small in comparison with other blocks of the code that run most of the time of the exploration, such as the code for simplifying strings or for generating children. This approach still does not fully eliminate the repetitive computations. It can still happen that two threads start to compute the outcome of the same couple $(P + n)$ if the nimber of P is not stored in the global database. In the case that $(P + n)$ is losing, both threads complete the computation, the first thread stores the computed nimber while the second thread finds out that the computed nimber is already in the database.

To deal with these kinds of re-computations, we implemented another synchronizing mechanism that allows threads to lock a couple before exploring it. Then, each thread checks whether a couple is not locked by another thread before exploring the couple. If it is, it can work on other positions in the meantime and pick up the computed result of the locked couple later. This approach highly improves the performance on positions that previously had a large number of re-computations. It also has an insignificant overhead, and thus the performance on other positions remains almost the same.

We measured the computation times of GLOP and our program using a single and four threads to compare the performance; see Figure 2.16. The computation of outcomes with our program is slightly faster than with *GLOP*, even when using a single thread. With four threads, the computation can speed up significantly.

$P_{7,4}$	`0*3.ABCDEFGH|ABCDEFGH`
$P_{7,6}$	`0.ABCDEFGHIJKL|ABCDEFGHIJKL`
$P_{8,2}$	`0*4.1a1a.AB|0.AB`
$P_{8,4}$	`0*2.ABCDEFGH|0*2.ABCDEFGH`
$P_{8,6}$	`0.1aABCa.DE|1a1a.DE|1ABC`
$P_{9,2}$	`0*5.1a1a.1a1a`
$P_{9,4}$	`0*3.11a1a.1abc1cba`
$P_{9,6}$	`0*3.ABCDE|1aA1BCD1Ea`
$P_{10,2}$	`0*6.1a1a.AB|0.AB`
$P_{10,4}$	`0*4.1aAa|0*2.1abAba`
$P_{10,6}$	`0*2.1a1a.1abAba|0.1abcAcba`

Table 2.5: The randomly selected positions $P_{i,j}$ that we used to measure the computation times in Figure 2.16.

Despite all the previously proposed methods, the distributed computation has not improved the performance as much as we expected. The average computation time did not decrease proportionally to the number of threads. On the contrary,

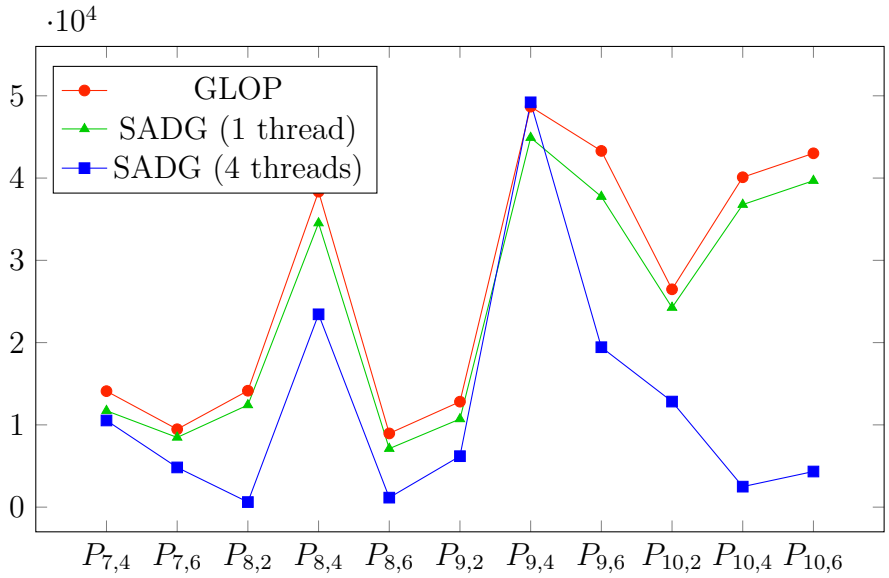


Figure 2.16: Comparison of the computation times of *GLOP* and our program (SADG). Every column $P_{i,j}$ contains the average times (in milliseconds) of determining the outcome of 3 games starting on the i-spot position $P_{i,j}$ with j randomly selected moves with empty databases of pre-trained moves. The positions $P_{i,j}$ are listed in Table 2.5. All the computations were performed on a computer with the Intel(R) Core(TM) i7-6700HQ CPU running at 2.60GHz with 16GB of RAM.

using too many threads slowed down the performance in most cases. It is quite surprising since the synchronization between threads is minimal, and thus the acceleration could be approximately proportional to the number of used threads. In our opinion, this behaviour is mainly caused by an extremely large number of dynamic allocations and deallocations during the computation which results in the overwhelmed heap allocator that prevents any further acceleration by using more threads. This problem could be resolved by reducing the number of allocations as much as possible. We were not able to do that in a simple way since the design of our program highly depends on the dynamical allocations.

The improvements of Sprouts solvers by using distributed computations are still opened for a future work. Such an implementation might use similar approaches to those proposed in this subsection and in our opinion, it should be designed so that it will contain a minimum number of allocations.

We do not use distributed computations directly in our game since it behaves relatively unpredictably on different devices or even on a single device as one can see in Table 2.16. Furthermore, using distributed computations leads to a non-deterministic exploration of a game tree, and thus it is not suitable for creating a computer opponent based on pre-trained databases since the exploration of a game tree in the game must match with the one used during the training.

2.3.6 Training of the computer opponent

We stored the computed nimbers of positions in a database to avoid repetitive computations in Subsection 2.3.4. After a game tree exploration is completed, we keep the database by storing it into a file and we reuse it later in other

computations. We are thus able to equip our application with a strong database before releasing so that it will never take the perfectly playing opponent too long to find a best move. Such a database is called a *pre-trained database* and we denote it by D_p. In this subsection, we discuss how to generate such a database D_p together with configurations of the perfectly playing opponent that are used during a training.

First, consider an n-spot position P whose outcome is determined by Algorithm 8. The run of the algorithm stores all the computed nimbers into a database D. Then, if we run the same algorithm on P again with the database D with the previously computed nimbers, we will obtain the result immediately. This is true for a losing position P because the first call of the algorithm stores all the nimbers of lands of P. Then, all these nimbers can be reused in the second call of the algorithm to obtain the immediate result. For a winning position P, the nimber of the land that forms the couple $(P' + n')$ is not stored as it is a winning couple. However, there exists a losing child of $(P' + n')$ that consists only of lands whose nimber is known, and thus the outcome of $(P' + n')$ is immediately reported as the outcome of P.

Thus, if we want to have an assurance that the computation of the outcome of a position P will never take too long, we take the pre-trained database D_p and use it for the computation of the position P. We call this process a *training* of the position P in the database D_p. After the database D_p is extended by the nimbers that were computed during the computation of P, we say that P is *trained* in D_p. We use two more additional terms: the *full training* of a position P and a *fully trained* position P. Both terms correspond to the previous ones but we do not only train the position P but also all positions that can be obtained from P in a game between the perfectly playing computer player and a human opponent. Since the computer player can work with a different set of moves than the human opponent, we need to distinguish between the first and the second player in P. The computer player is the first player in P if and only if P is losing as we do not let the human opponent start in a losing position in our program.

Note that the training of P does not have to produce the same database D as the full training of P. It is caused by the fact that, unlike the full training of P, a training of P does not have to explore all the parts of the game tree of P since some explorations of the game tree could be pruned during the training by finding a losing child; recall Algorithms 6 and 8. Therefore, if we want to ensure that the perfectly playing opponent will never think too long on a game with a starting position P, then it is not sufficient to only train the position P, we need to fully train P. A good example is the training of the position `0*8` producing a database D that takes only about five seconds. Surprisingly, the training of the position P' with the string `0.ABCDEFGHIJKLMN|ABCDEFGHIJKLMN`, which can be obtained from `0*8`, together with using the database D of previously computed nimbers takes about half an hour. Even though P' cannot be obtained from `0*8` in a game between the perfectly playing computer opponent and a human player, the example nicely illustrates that difficult positions that are not explored during the training of P can occur in the game tree of a position P.

It could seem that we need to train each position occurring in the game tree of P to fully train P. However, not all positions of the game tree can be obtained in the game between the computer and a human player. A human opponent can

play any of the possible moves including moves that lead from a winning position to a losing position but this does not hold for the perfectly playing computer player. The perfectly playing computer opponent will never play a move leading from a winning position into another winning position. Therefore, during the full training tree exploration of P, if we meet a winning position on which the computer player is on the move, then we do not further fully train its winning children.

We can restrict the computer's choice of possible moves even more by choosing upper bounds k_L and k_W on the number of moves a computer can play. First, if the computer opponent is the first player in a winning position P_W then it does not matter which of the moves leading to a losing position it chooses since he will win anyway. Therefore, the computer player chooses an arbitrary child from the first k_L children of $\mathcal{P}_L$ ordered by the priorities from Subsection 2.3.4. This allows us to fix the computer's choice and it also makes the computer player to choose children that are simpler to analyze than others which improves the performance of the full training.

We can also apply this approach to losing positions since the computer player will lose anyway if the opponent plays perfectly. However, the human opponent often does not. Thus, it would be unreasonable to choose a winning child of a losing position P_L randomly. Instead, we assign a priority p to each winning child P_W of P_L expressing the probability that the human opponent playing on P_W makes a mistake by making a move leading from a winning position to other winning position. More precisely, if P_W has l losing and w winning children, then $p = \frac{w}{w+l}$ (note that $w + l > 0$ since P_W is winning). After the priorities are assigned, the computer player takes the first k_W winning children of P_L ordered by their priorities and then chooses one of them proportionally to their priorities.

The disadvantage of using these priorities is the fact that besides computing the outcomes of each winning child P_W of P_L, we need to also compute the outcome of each child of P_W to determine the values l and w. Another disadvantage is that the computer player does not use the priorities from Subsection 2.3.4, and thus it can potentially choose children that are difficult to analyze. These disadvantages prevent us to use the probabilities on positions with a high number spots. However, it is still possible to use them in most of the full trainings in our program. Moreover, on large positions it can take too much time to compute the probabilities during the game even though the probabilities were pre-trained. We solved this issue by considering at most 20 winning children of P_L with minimal priorities in the computations of probabilities. In the full trainings of large positions where the probabilities could not be used, the computer chooses the winning children of a losing position P_L randomly in the same way as in the selection of losing children of a winning position but it uses the constant k_W instead of the constant k_L.

There are no general values of constants k_L and k_W that we would use for all full trainings, and thus we must arrange that a computer player plays on a position P with the same configuration he was trained with on P. The higher the values of the constants are the more variously the computer player plays. On the other hand, with higher values of k_L and k_W the full training becomes more difficult. Therefore, we set the values of the constants k_L and k_W together with the decision of using probabilities p for a full training of each position separately so that it is still computable. In our program, we train positions that can be

obtained from an n-spot position with n up to 11 and their configurations are shown in Table 2.6.

n	k_L	k_W	p
≤ 6	∞	∞	✓
7	2	5	✓
8	2	4	✓
9	2	2	✓
10	1	2	✓
11	1	2	✗

Table 2.6: The values of k_L and k_W used for the training of a position P according to the smallest n-spot position from which P can be obtained. The values in the last column mark whether we use the priorities p (✓) or not (✗).

The outlined problems show that it is considerably more difficult to implement a computer player that plays perfectly on a position P than to determine the outcome of P since we need to explore the game tree of P in a much larger width during the full training. Therefore, it is naive to expect that we would be able to implement a computer opponent that would be capable to play perfectly on similarly complex positions as GLOP is able to analyze. Recall that GLOP manages to compute the outcomes of n-spot positions for n up to 53 [6]. However, by modifying the computer's choice of possible moves, we can reduce the size of the explored game trees, and thus our computer opponent can play perfectly on larger positions than any existing Sprouts AI does.

2.4 Synchronization of the representations

In Section 2.1, we described the graphical representation $gr(P)$ to capture the shape of a position P and then, in Subsection 2.3.1, we discussed the string representation $sr(P)$ of a position P that allows us to find a best move in P. Now, we introduce methods how to synchronize these two representations so that it is possible to draw a computer move found with the string representation into the graphical representation. This step is probably the most difficult obstacle we had to deal with and, as far as we know, a full analysis of the computer opponent's drawings is missing in the literature. Therefore, we come up with our own solution that is based on *Delaunay triangulations*, where the most important step is called the *spindle method*.

First, we need to determine which move should be drawn in the graphical representation so that it will correspond to the string chosen by the computer opponent. Thus, we describe the graphical representation $r(m)$ of a move m together with the ways how to synchronize it with the string representation in Subsection 2.4.1.

Then, the move m has to be drawn into the graphical representation according to its representation $r(m)$. This step is the core of the synchronization and it is described in detail in Subsection 2.4.2.

It seems that we could use a standalone graphical representation and the string representation would be only created on demand from the graphical representation. This approach is possible and it does not require any further synchronization since the string representation would be discarded after each use. However, in Subsection 2.3.5, we will see a disadvantage of this approach which leads to a necessity of using both representations in parallel. Since both representations are used during the whole game, each modification of a representation must be projected into the other one which results in further synchronizations described in this subsection.

2.4.1 Move graphical representation

Consider a position P represented by the graphical representation $gr(P)$ and by the string representation $sr(P)$. If the computer opponent chooses a child string $sr(P')$ of $sr(P)$ as the best move, then the goal is to determine the representation $r(m)$ of a move m that should be drawn and inserted into $gr(P)$ so that its string representation will be equivalent to $sr(P')$.

First, we characterize the move m by the move representation $r(m)$ so that it will be later possible to draw it into $gr(P)$ unequivocally. In $r(m)$, we store vertex occurrences which are connected by m. Since single-boundary moves split the surrounding boundaries $gr(\mathcal{B})$ into the major partition $gr(\mathcal{B}_{major})$ and the minor partition $gr(\mathcal{B}_{minor})$, we store $gr(\mathcal{B}_{major})$ in $r(m)$. We do not store $gr(\mathcal{B}_{minor})$ as it can be obtained as the complement of $gr(\mathcal{B}_{major})$ in $gr(\mathcal{B})$. The representation $r(m)$ additionally contains the reference to the region $gr(R)$ in which the move m is done because then we do not have to find $gr(R)$ using to the connected boundaries.

It is not possible to construct the move representation $r(m)$ only with the information from the string representation since it does not contain any references to the corresponding graphical representation. We could try to pair each boundary $sr(\beta)$ with the corresponding boundary $gr(\beta)$ but it would be extremely difficult as the number and the order of vertices can substantially differ after the simplification of $sr(\beta)$.

Therefore, when the string representation $sr(P)$ is created from the graphical representation $gr(P)$, we extend $sr(P)$ by adding necessary references to $gr(P)$ that are kept during the simplification and also during the generating of the children. We keep these references only for the first generation of children when we are exploring the game tree to find a best move.

There are three types of references: for regions, boundaries, and for vertex occurrences. Each region $sr(R)$ contains a reference to the corresponding region $gr(R)$. It would be natural if each boundary $sr(\beta)$ would also contain a similar reference to $gr(\beta)$. However, Step 5 of the string reduction can merge up to three different alive boundaries into a single one. Therefore, we do not store the references in the boundaries but we store them later for vertices, and thus each vertex $sr(v)$ contains a pair of references $(i, gr(\beta))$ for the ith vertex occurrence in the boundary $gr(\beta)$.

Now, after two vertex occurrences in a boundary $sr(\beta)$ are connected, we are able to identify the corresponding graphical occurrences and the corresponding surrounding region. For a single-boundary move, we further need to fill the major

partition $gr(\mathcal{B}_{major})$. When a string of a single-boundary child is generated, we work with its major and minor partitions $sr(\mathcal{B}_{major})$ and $sr(\mathcal{B}_{major})$. So we could simply add the referenced boundaries from $sr(\mathcal{B}_{major})$ into $gr(\mathcal{B}_{major})$. However, the string pseudo-canonization of $sr(\beta)$ can cause that the partitions of a single-boundary move on $sr(\beta)$ will not correspond to the partitions of the corresponding move on $gr(\beta)$. In both $sr(P)$ and $gr(P)$, the major boundary is formed by the occurrences with increasing indices between the occurrences of the endpoints of the move. Nevertheless, after $sr(\beta)$ is rotated or reversed during the pseudo-canonization, it can happen that the endpoint occurrence with the smaller index is referencing to the endpoint with the higher index. Then, the decision whether the partitions correspond depends on the order of the referenced endpoints and the orientations of $sr(\beta)$ and $gr(\beta)$; see Algorithm 10.

Algorithm 10: Check if the partitions of a single-boundary move on a boundary $sr(\beta)$ correspond.

```
Function DoesPartitionsCorrespond(sr(β))
    sr(A_i) ← the first endpoint occurrence;
    sr(A_j) ← the second endpoint occurrence;
    if i = j then
        return True;
    gr(A_i') ← the referenced occurrence to sr(A_i);
    gr(A_j') ← the referenced occurrence to sr(A_j);
    gr(β) ← the referenced graphical boundary to sr(β);
    if the orientations of sr(β) and gr(β) are equal then
        return i' < j';
    else
        return j' < i';
```

Now, when the computer opponent chooses a best child $sr(P')$ of $sr(P)$, we can fill the move representation $r(m)$ that is used to draw the corresponding move m to $gr(P)$. Note that since the string reduction can merge boundaries together, it can happen that the move forming $sr(P')$ is a single-boundary move whereas the move drawn in $gr(P)$ is a double-boundary move. This does not affect the game tree of P though, as it is preserved by the merge boundaries operation.

2.4.2 Drawing a computer move

The goal of this subsection is to draw a move m according to its representation $r(m)$ into the graphical representation $gr(P)$. Drawing such a move m is the most complicated part of the synchronization since there are many factors that need to be considered mainly during drawing single-boundary moves. Furthermore, we require that the shape of m is natural to resemble a move drawn by a human player.

Proposed solutions Before we introduce our methods for drawing moves, we mention the solutions proposed in the literature. In fact, we are aware of only

one paper analyzing drawings of a computer opponent. In his paper about optimizations for generating children of a position, Browne [5] sketched out the idea of visualizing Sprouts moves by using *Delaunay triangulations* and *Voronoi diagrams*. Even though the introduced solution would work relatively well for the n-spot positions, we doubt that the algorithm would work for more complicated positions. Besides many unexplained parts of the algorithm, the main problem is that it does not distinguish the major and minor partitions of a single-boundary move and it does not take into account the border boundary of a region in which the move is drawn. These two issues do not have to be considered in the n-spot positions since the move is made in the outer region and the partitions of single-boundary moves on singletons are interchangeable. However, both of the problems need to be explicitly solved in more general positions since otherwise we could draw a move m not corresponding to its representation $r(m)$. Figure 2.17 shows two non-equivalent moves that differ only in the swapped partitions.

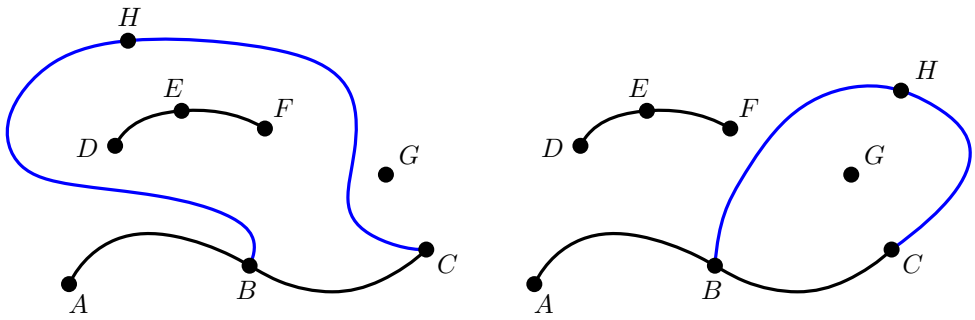


Figure 2.17: An example of two single-boundary moves leading to non-equivalent positions that differ only in swapping the major and the minor partition.

Triangulations Similarly to Browne [5], we use *constrained conforming Delaunay triangulations* to draw a move. A *Delaunay triangulation* is a triangulation of a planar set P of points in general position such that no point of P is inside the circumcircle of any triangle of the triangulation. A *constrained Delaunay triangulation* of a face of a rectilinear plane graph G is a generalization of the Delaunay triangulation that can force the triangulation to use some edges of G called the *constrained edges*. Finally, the *constrained conforming Delaunay triangulation* (CCDT) [32] is a constrained Delaunay triangulation using *Steiner points* which are points added to the triangulated set to meet given constraints on the minimum angle and the maximum area of the triangles. We apply these constraints to improve the overall quality of the triangulations.

For example, we use the CCDT to triangulate a region whose edges are all constrained; see Figure 2.18. To compute the CCDT in our program, we use the library *Triangle.NET* [33], which is a C# port of Shewchuk's *Triangle* software [32].

Searching a path inside a triangulation During drawing of a computer move, we often need to find a path between two points p_1 and p_2 of a triangulation $\mathcal{T}$

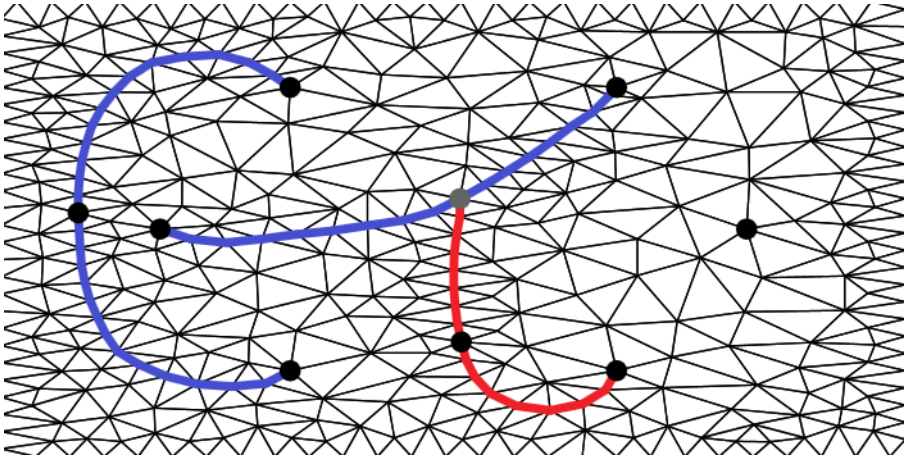


Figure 2.18: An example of a triangulated region by the CCDT [33] with the constraints set to the minimum angle 30° and the maximal area 0.005 and with an unlimited number of Steiner points.

where p_1 and p_2 are not the Steiner points of $\mathcal{T}$. To do so, we define a plane graph $G_\mathcal{T} = G_\mathcal{T}(p_1, p_2)$ over the triangulation $\mathcal{T}$. The vertices of $G_\mathcal{T}$ are the vertices p_1, p_2, and the midpoints of all non-constrained edges of $\mathcal{T}$. Two vertices are connected by an edge in $G_\mathcal{T}$ if they lie in the same triangle of $\mathcal{T}$ but not on the same edge of $\mathcal{T}$. If p_1 and p_2 lie in the same triangle T, then we add the center of gravity c of T into $V(G_\mathcal{T})$ and edges p_1c and cp_2 into $E(G_\mathcal{T})$. We obtain the following result from the definition of $G_\mathcal{T}$.

Observation 13. *Given a triangulation $\mathcal{T}$, two non-Steiner points p_1, p_2 of $\mathcal{T}$, and the graph $G_\mathcal{T}$, then p_1 and p_2 are reachable by a path in $G_\mathcal{T}$ and no path between points p_1 and p_2 crosses any constrained edge of $\mathcal{T}$.* □

To find the shortest path between p_1 and p_2, we simply use the *breadth-first search* (BFS). We could also use the *Dijkstra's algorithm* instead of BFS to take into account the lengths of edges of $G_\mathcal{T}$. However, it is not necessary since our triangulations are fine enough, and thus the paths are not too long.

We often use a modification of the BFS that searches for the shortest path between any two not necessarily disjoint sets of non-Steiner points of a triangulation $\mathcal{T}$. Since the endpoints of the path are often game vertices and the sought path corresponds to a drawn move, we improve the algorithm so that it allows to search for a path between two occurrences A_i and B_j of game vertices. Therefore, the algorithm is equipped with the function from Subsection 2.7 that checks whether a small edge is correctly connected to an occurrence. Then, the BFS considers only paths that are correctly connected to the occurrences A_i and B_j. There are two shortest paths between the same game vertices but different occurrences in Figure 2.19; recall that these moves do not have to be equivalent.

Drawing a double-boundary move Drawing a double-boundary move between two occurrences A_i and B_j in a region R is now trivial as we only need to construct a triangulation $\mathcal{T}$ of the region R and find the shortest path in $G_\mathcal{T}(A_i, B_j)$ between A_i and B_j.

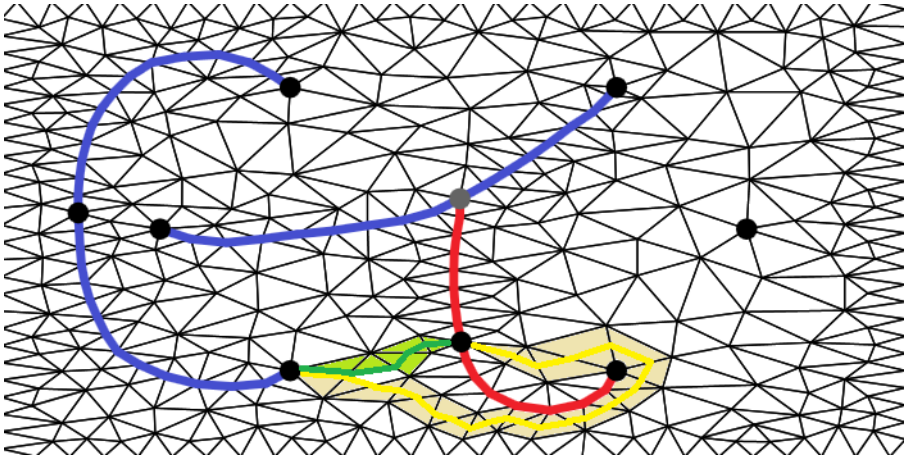


Figure 2.19: An example of two paths in a triangulation that connect different occurrences of the same vertices.

Drawing a single-boundary move Consider a single-boundary move m connecting occurrences B_i and B_j with $i \leq j$ on a boundary β of a region R with a set $\mathcal{B}$ of boundaries that splits $\mathcal{B} \setminus \{\beta\}$ into a major partition $\mathcal{B}_{major}$ and a minor partition $\mathcal{B}_{minor}$. Drawing of m is much more complicated than drawing of a double-boundary move since we have to deal with the freely lying boundaries that are split into two partitions.

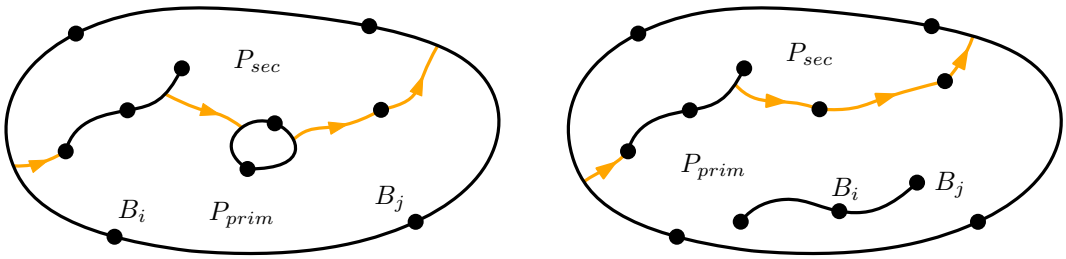
Therefore, the first step of our algorithm is to connect all the inner boundaries from $\mathcal{B} \setminus \{\beta\}$ by a curve called *spindle*[2] that starts and ends in the border boundary (or the border of the playing area if R is the outer region); see Figure 2.20. We use the term border boundary also for the border of the playing area. Then, we intertwine m with the spindle so that the partitions $\mathcal{B}_{major}$ and $\mathcal{B}_{minor}$ are on the correct sides of m; see Figure 2.21. Intertwining m correctly requires a lot of technical steps that are described in detail below.

Setting up the spindle The spindle starts at an arbitrary occurrence of the border boundary and leads to the closest vertex of an inner boundary from $\mathcal{B} \setminus \{\beta\}$. Then, the spindle continues from a vertex of the last visited boundary to the closest vertex of a non-visited inner boundary from $\mathcal{B} \setminus \{\beta\}$. We continue in this manner until we visit all the inner boundaries from $\mathcal{B} \setminus \{\beta\}$. Then, we end the spindle by connecting it to the closest occurrence of the border boundary that differs from the starting occurrence. If there is no first inner boundary that could be connected by the spindle, we simply lead the spindle between two different occurrences of the border boundary. It then does not matter which occurrences are chosen since the spindle is not used in this case.

Each part of the spindle is constructed using a triangulation $\mathcal{T}$ of the gradually modified region R and the path-searching algorithm over the graph $G_{\mathcal{T}}(B_i, B_j)$. We have one more requirement for the starting and the ending occurrence of the spindle: if β is the border boundary, which means that the move m is made there, then the starting and the ending occurrence cannot be equal or lie between the occurrences B_i and B_j. After the spindle is finished, the border polygon of R is divided into the two following polygons. The *primary polygon* P_{prim} is the polygon

[2]We use the term spindle since it is similar to the mitotic spindle during a cell division.

that contains β or the occurrences B_i and B_j if β is the border boundary. The other polygon is called the *secondary polygon* P_{sec}. Finally, we orient the spindle so that its orientation is opposite to the orientation of the spindle induced by the counterclockwise orientation of P_{prim}. There are two examples of spindles in Figure 2.20. One spindle is constructed with a border boundary move and the other one with an inner boundary move.



A border boundary move. An inner boundary move.

Figure 2.20: Examples of spindles (denoted by orange curves) that split the region into P_{prim} and P_{sec}.

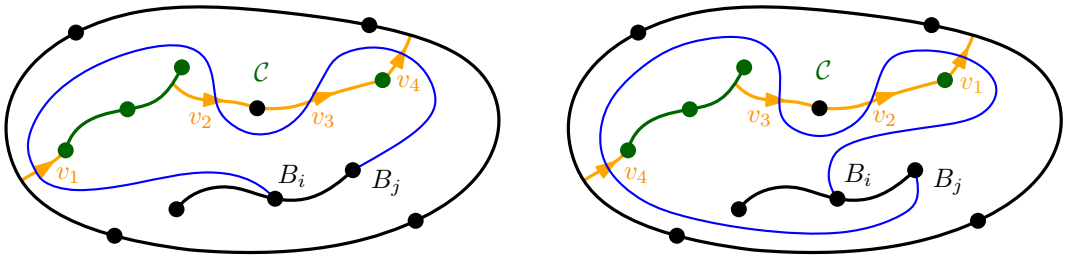
Intertwining the spindle After the inner boundaries from $\mathcal{B} \setminus \{\beta\}$ are visited by the spindle s, we start intertwining the move m to partition the boundaries. Let $\mathcal{C}$ be a subset of the set of inner boundaries of R visited by the spindle. To describe the intertwining, we introduce a process of *enfolding* $\mathcal{C}$. The boundaries from $\mathcal{C}$ are then called the *enfolded boundaries*.

The enfolding starts with drawing a curve in P_{prim} from the occurrence B_i to a vertex v_1 of the first segment of the spindle s that precedes a boundary from $\mathcal{C}$ with respect to the orientation of s. Then, we continue in P_{sec} by drawing a curve from v_1 to a vertex v_2 of the first segment of s that precedes a boundary not in $\mathcal{C}$ and then we continue in v_2. We repeat this process until we get at the end of the spindle. In this moment, if we should continue drawing in P_{sec}, we just draw a curve to the last segment of s to get back to P_{prim}. Finally, we draw a curve from the last connected vertex to the occurrence B_j.

If $\mathcal{C}$ is empty, we draw a curve in P_{prim} from B_i to the first vertex of the last segment of s, then a curve in P_{sec} to the last vertex of the last segment of s, and then we finally return to B_j in P_{prim}. For now, this step is necessary for the correctness of the algorithm, but we will optimize it later.

We also define an analogous process that gradually draws a move from B_i to B_j called the *reversed enfolding* of boundaries from $\mathcal{C}$. The only difference between the enfolding and the reversed enfolding is that the latter one intertwines the spindle s in the opposite direction.

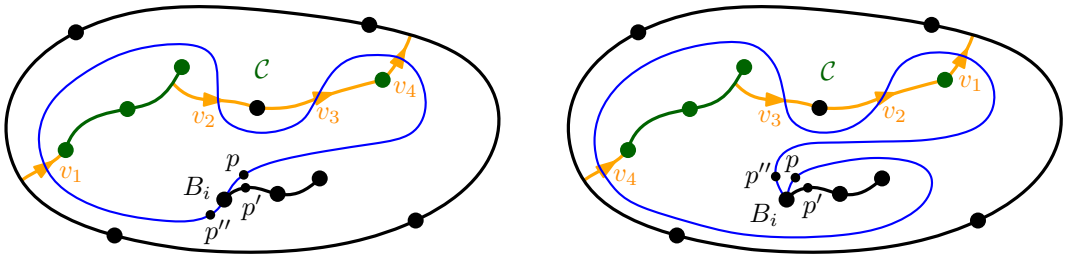
See Figure 2.21 for an example of the enfoldings. Note that if $B_i \neq B_j$, then the shape of the last curve does affect the properties of the move. However, this does not necessarily hold if $B_i = B_j$ since the last curve can be connected to B_i in two non-equivalent ways. In this case, we insist on the last curve c being connected to B_i so that the last drawn point p of c lies on the right side of the triple (p', B_i, p'') where p' is the neighboring point of B_i on β and p'' is the first



The enfolding of $\mathcal{C}$. The reversed enfolding of $\mathcal{C}$.

Figure 2.21: The enfoldings of the boundaries from $\mathcal{C}$ if $B_i \neq B_j$.

drawn point of the first curve; see Figure 2.22. We can determine the side of the connection using Table 2.1 from Subsection 2.1.3.



The enfolding of $\mathcal{C}$. The reversed enfolding of $\mathcal{C}$.

Figure 2.22: The enfoldings of the boundaries from $\mathcal{C}$ if $B_i = B_j$.

Note that the reversed enfolding cannot be always performed if the move is made on the border boundary. Nevertheless, we do not need it in this case. Similarly, we can see that the connection of the last curve in the case $B_i = B_j$ on a border boundary is unambiguous without using the extended definition. Again, to draw the curves, we use the triangulation $\mathcal{T}$ of the gradually modified polygons P_{prim} and P_{sec} and the path-searching algorithm over the graph $G_{\mathcal{T}}$ together with the correctness checks of the connections to occurrences.

Choosing the right enfolding Now, we use the enfoldings to draw a move m using its representation $r(m)$. We have four options how to enfold the partitions. We can choose $\mathcal{C}$ as the set of inner boundaries from $\mathcal{B}_{major}$ or from $\mathcal{B}_{minor}$. We can also apply either the enfolding or the reversed enfolding.

It follows from the definition of enfoldings that the enfolded boundaries $\mathcal{C}$ always lie in the major region of the move m whereas the reversely enfolded boundaries $\mathcal{C}$ always lie in the minor region. Therefore, we are left with only two options: we either enfold the inner boundaries from $\mathcal{B}_{major}$ or we reverse enfold $\mathcal{B}_{minor}$.

Recall that $\mathcal{B}_{major}$ and $\mathcal{B}_{minor}$ can contain the border boundary which cannot be enfolded nor reversely enfolded. Hence, the partition that contains the border boundary can never be enfolded or reversely enfolded since the border boundary would be separated from the inner boundaries in the same partition by m. Thus, we are now left with a single option for enfolding. We enfold $\mathcal{C} = \mathcal{B}_{major}$ if $\mathcal{B}_{major}$

does not contain the border boundary and we reverse enfold $\mathcal{C} = \mathcal{B}_{minor}$ otherwise. Compare the correct enfolding of boundaries from Figure 2.23 with the incorrect enfoldings of boundaries from Figure 2.24. We can see that each of the four possible enfoldings can lead to a different position.

Since the border boundary is not contained in $\mathcal{B}_{major}$ nor in $\mathcal{B}_{minor}$ during a border boundary move, the reversed enfolding is not necessary for a border boundary move.

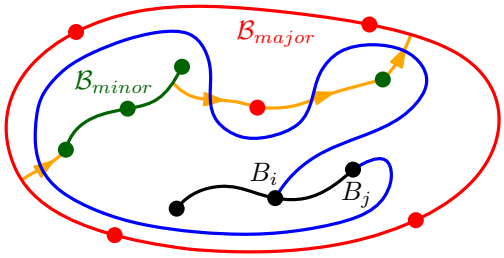
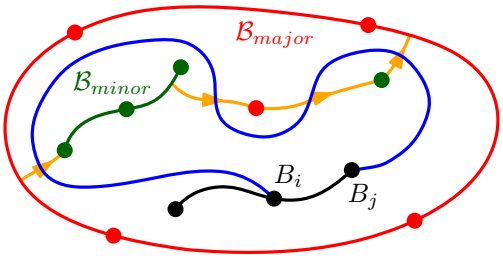
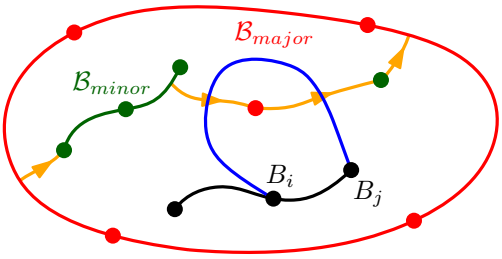


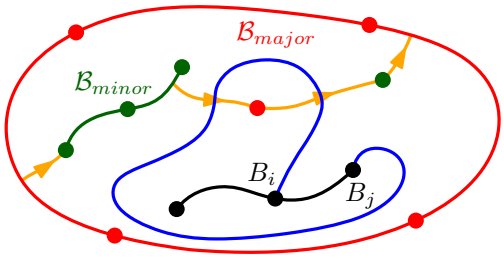
Figure 2.23: The correctly chosen reversed enfolding of $\mathcal{B}_{minor}$.



The enfolding of $\mathcal{B}_{minor}$.



The enfolding of $\mathcal{B}_{major}$.



The reversed enfolding of $\mathcal{B}_{major}$.

Figure 2.24: The incorrectly chosen enfoldings that draw a move m not corresponding to its representation $r(m)$ with the partitions $\mathcal{B}_{major}$ and $\mathcal{B}_{minor}$.

Choosing a smaller enfolded set During drawing of a single-boundary move, we choose between the enfolding and the reversed enfolding depending on which partition contains the border boundary. However, it can happen that the border boundary is missing because it is either dead or the surrounding region is the outer region. In this situation, we can freely choose between the enfolding of the major partition and the reversed enfolding of the minor partition. Since one move can be shorter than the other one, we measure the length of each boundary in a partition and enfold the partition with a smaller sum of lengths. In these

measurements, we do not count the lengths of dead boundaries since they are ignored during enfolding. In general, this approach is sufficient to find the shorter move and obtain more natural drawings.

Furthermore, we use the fact that the singletons are mutually interchangeable. Therefore, if n singletons should be enfolded during the drawing, we are allowed to choose an arbitrary set of n singletons on the spindle that will be enfolded. We enfold the n singletons that are the closest to the to the point between the connected occurrences. This approach is good enough for positions with a small number of singletons. However, on positions with a higher number of singletons it is not unusual that a chosen set of singletons leads to unnecessarily long move. This behaviour is caused by the fact that the closest singletons are relatively spread out over a long spindle, and thus the move must go through a large part of the spindle. Hence, the approach of choosing the set of enfolded singletons could be improved in a future work since some single-boundary moves on positions with a large number of singletons look a bit unnatural.

Optimizing empty single-boundary moves In the paragraph about single-boundary moves, we used the fact that a move is always drawn across the last segment of a spindle when no boundary was enfolded. We introduced this method to fix the orientation of the drawn move, and thus to determine the partition in which the un-enfolded boundaries will lie depending only on the direction of enfolding and not on the shortest move in which the primary polygon P_{prim} is found. However, these type of moves seem unnatural since they are drawn to a certain part of the region and then they return back without enfolding anything. We optimize this approach now but it needs some effort since in some cases we need to fix the orientation of the drawn move.

We start with the most simple moves that do not enfold any boundaries and then we describe one more advanced case. The moves on the border boundary are the simplest. In this case, we simply find the shortest path in the triangulation $\mathcal{T}$ of P_{prim} since all these moves always form the empty major region that does not contain any boundary on the spindle. The only exception are empty loops connecting the same occurrences. We solve this case in the same way as the empty loops on singletons. We find a first triangle in $\mathcal{T}$ containing the occurrence and then we pick two points inside the triangle that form the loop. This simple method creates very small moves but they will be stretched out by the redrawing algorithm. We do not care about their orientation since all empty loops on a singleton or on a border boundary are equivalent.

The empty move on a general inner boundary β connecting occurrences B_i and B_j, where $i \leq j$, is more advanced since the orientation of the move decides whether the boundaries on the spindle lie in the major or in the minor region. Therefore, we enforce the correct orientation by connecting β to the primary polygon P_{prim} using a so-called *splitting curve*. After the splitting curve is connected, the primary polygon P_{prim} is modified so that there remains only one possible orientation of a move connecting the corresponding occurrences of B_i and B_j on P_{prim}.

Now, we distinguish the cases $B_i \neq B_j$ and $B_i = B_j$. If $B_i \neq B_j$, we connect the splitting curve from the border of P_{prim} to any occurrence between B_i and B_j if the boundaries on the spindle and the border boundary (if it exists) are from $\mathcal{B}_{major}$. Otherwise, if the boundaries are from the minor partition $\mathcal{B}_{minor}$, we

connect the curve to any occurrence from the complementary range except of the occurrences B_i and B_j themselves. Then, we simply draw a move in the modified polygon P_{prim} between B_i and B_j.

If $B_i = B_j$, then we connect the splitting curve from the border of P_{prim} to the occurrence B_i if the boundaries on the spindle and the border boundary (if it exists) are from $\mathcal{B}_{major}$. Then, we make a move from and to the vertex behind the occurrence B_i so that the starting occurrence on P_{prim} differs from the ending occurrence on P_{prim}. If the boundaries are from the minor partition $\mathcal{B}_{minor}$, we simply make an empty loop using the previously mentioned approach. See Figure 2.25 for the more advanced empty moves.

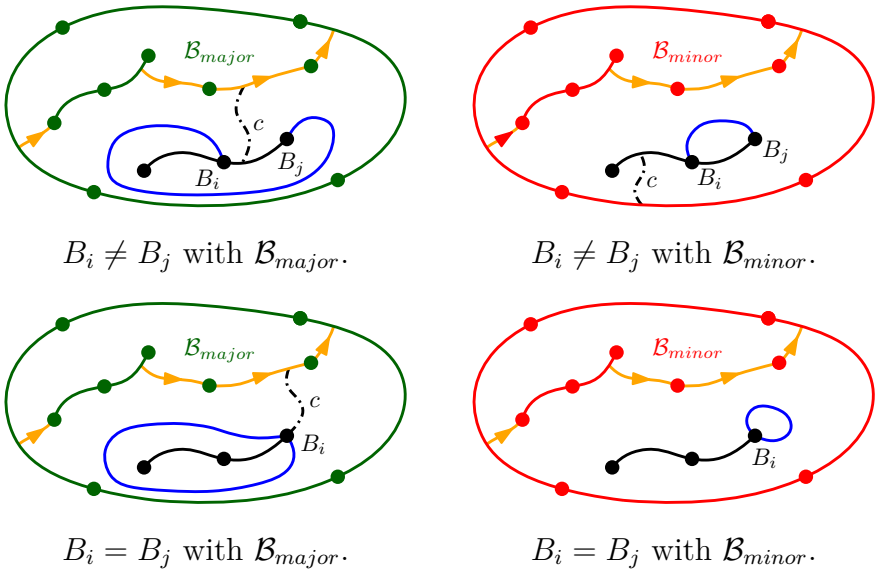


Figure 2.25: Examples of moves that do not enfold any boundaries on the spindle and that often need to use the splitting curve c.

Redrawing a move The drawn moves are often too close to other edges and they are also often relatively spiky as underlying triangulations are often not fine enough. To solve this and to make the drawn moves smoother and nicer, we use the redrawing algorithm from Section 2.2 just before the drawn edge is animated. We use a modification of the redrawing algorithm that moves only the inner vertices of the drawn edge.

2.4.3 Parallel representations

As we have motioned, it is possible to maintain a standalone graphical representation and use the string representation only on demand to determine a computer's best move. This approach is simple because the conversion from the graphical representation $gr(P)$ to the string representation $sr(P)$ is trivial since $gr(P)$ is only an extension of $sr(P)$ (except of the lands but they can be simply determined using Algorithm 4). After the move is found, $sr(P)$ can be discarded and therefore, any modification of $gr(P)$ or $sr(P)$ does not have to be projected to the other one.

However, a disadvantage of this approach is that there are multiple ways of the conversion from $gr(P)$ to $sr(P)$ which can lead to different but equivalent strings. This causes a problem with the pre-trained database because when we fully train a position $sr(P)$ we assume that each position in a game G between the computer opponent and a human player on $sr(P)$ is trained in the database. However, it can happen that the converted string cannot be reached in G, and thus it is not trained in the database and we have no assurance that we compute the position immediately.

Therefore, we are forced to synchronize two parallel representations $gr(P)$ and $sr(P)$ representing the same position P. Then, if a human player makes a move, the move is inserted into $gr(P)$ and $sr(P)$ is updated to one of its children corresponding to the move. In the opposite direction, if the computer player chooses a best child of $sr(P)$, the corresponding move is drawn in $gr(P)$. If we have had used this two-way synchronization, we would have to distinguish between a move inserted into $gr(P)$ by the human player and by the computer player since only the first one leads to updating $sr(P)$. Instead, we always update $sr(P)$ after a move is inserted into $gr(P)$ and we do not update it after the computer opponent chooses one of its children.

Now, we are facing three tasks. First, $sr(P)$ must be updated by its corresponding child after a human player inserts a move into $gr(P)$. Second, when a computer player inserts a move into $gr(P)$, it is not enough to find a corresponding child but we must find the exact child on which the move was drawn since choosing a different, but equivalent, one could again give an un-trained position as the computer player does not have to consider the full set of possible moves. The third task is to update all the references from $sr(P')$ to $gr(P')$ after $sr(P)$ is updated to the child string $sr(P')$,

We have successfully implemented all the three tasks, and thus the computer opponent will never take too long to find a best move on a fully trained position in our program. The synchronization is complicated by the string simplification which causes the boundaries in $sr(P')$ to not directly correspond to the boundaries of $gr(P')$ and, even worse, due to the step of merging boundaries, a single boundary in $sr(P')$ can correspond to more boundaries in $gr(P')$. However, since the solution consists of a relatively large number of simple steps that highly depend on the implementation details, we have decided to not explicitly describe them in detail here.

3. Implementation

In Chapter 2, we discussed all the algorithms and data structures that deal with the three main programming challenges of the Game of Sprouts. In this chapter, we describe their implementation in our application *Sprouts: A Drawing Game* together with the used technologies and the overall structure of the solution. All the described files are listed in Attachment A.

3.1 Used technologies

Before describing the overall structure of the solution, we need to make some decisions on the used technologies. To implement the program, I decided to use the programming language C# along with the framework .NET Core 3.1 because it is the language I have the most experience with. The advantage of C# is that it is a simple and widely used language, and thus it will be easier for other developers to read and to possibly extend the program. Furthermore, .NET supports many platforms, and thus, even if we are mainly targeting Windows, it will not be difficult to extend the application to other platforms, such as the Android mobile devices, in the future.

In Goal **G4**, we decided to implement a user-friendly GUI. Note that Sprouts has a very simple graphics. We only need to draw elementary 2D objects such as circles, line segments, and some kind of spline curves which, moreover, change their positions and shape only on demand. Therefore, since we do not have any high requirements for the potential GUI framework, we do not have to use any complex game engines such as *Unity* and we can make do with simple ones. Since we are targeting Windows desktops, we decided to use the popular *Windows Presentation Foundation* (WPF) framework together with the *Material Design Themes* [34] package giving the WPF controls modern-looking styles.

Besides the standard libraries, we also use two external libraries in our program. The first one of them is the already mentioned *Triangle.NET* [33] library, which is a C# port of Shewchuk's *Triangle* software [32]. It is a library with a relatively simple interface for generating 2D (constrained) Delaunay triangulations and high-quality meshes of point sets or planar straight line graphs, which is exactly what we needed to draw a computer move as described in Subsection 2.4.2. The second external library is *Newtonsoft.Json* [35] that allows to convert between .NET objects and JSON. In Goal **G5**, we decided to support remote games and, in Goal **G6**, we promised to enable storing custom positions. Both of these goals need to serialize or deserialize .NET objects in a certain moment, for which we exploit the Newtonsoft.Json library.

3.2 Solution structure

Now, we describe the overall structure of the implementation Sprouts: A Drawing Game that was developed in *Visual Studio Enterprise 2019*. The implementation consists of a single solution `Sprouts` that is divided into four projects `GameModel`, `NetworkClasses`, `Server`, and `UserInterface`. The projects together with their

external dependencies are shown in Figure 3.1.

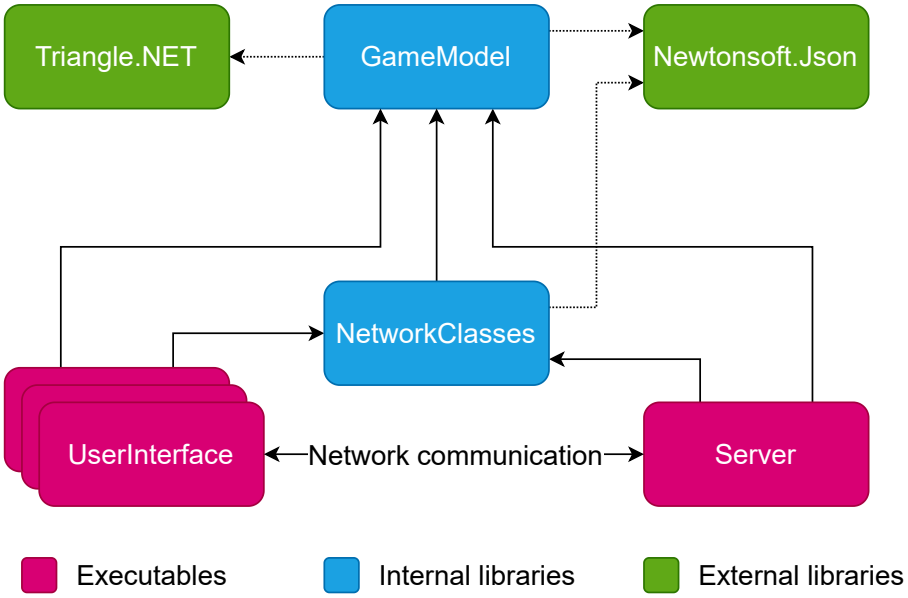


Figure 3.1: The structure of the `Sprouts` solution.

`GameModel` is a core library of the solution that primarily holds the state of a Sprouts position. It implements all the data structures and the algorithms that were described in Chapter 2. Since it contains the algorithms that draw a computer move, it depends on the external library Triangle.NET [33]. It also has a reference to Newtonsoft.Json because the objects in `GameModel` are often serialized.

`NetworkClasses` is a small library containing simple classes that are shared between `UserInterface` and `Server` during the network communication. It also contains a general method that serializes and deserializes .NET objects with default settings for the whole application, and thus it depends on Newtonsoft.Json. Since the shared data contains the graphical representation, it has a dependency on `GameModel`.

`Server` is a console application that coordinates the communication between individual clients represented by the application `UserInterface` during a remote game. It uses the classes from the `NetworkClasses` library since it is one of the endpoints of the network communication. It also depends on `GameModel` because it sometimes explores the sent data to determine the state of a game.

`UserInterface` is a WPF application producing the resulting executable file `Sprouts - A Drawing Game.exe`. Besides the XAML documents designing the appearance of the user interface, it also contains controllers defining the behavior of the individual game modes. The project includes further functions processing game canvas drawings together with functions that display positions stored in `GameModel`. Finally, the project `UserInterface` also contains a network client module that communicates with `Server` during a remote game, and thus it depends on `NetworkClasses`.

The solution also contains the `Setup` project which is the part of the official Microsoft extension [36] of Visual Studio that produces installers of projects. We apply it to create an installer of the `UserInterface` project. Since the setup project is very simple, we only describe the resulting installer in Subsection 4.1.1.

3.3 GameModel project

The project `GameModel` is the core of the application that contains all the important data structures and algorithms from Chapter 2. This project is divided into four directories `DrawingAlgorithms`, `DrawingStructure`, `CanonicStructure`, and `Solver`. The overall structure of the project and the mutual dependencies between the most important classes are shown in Figure 3.2.

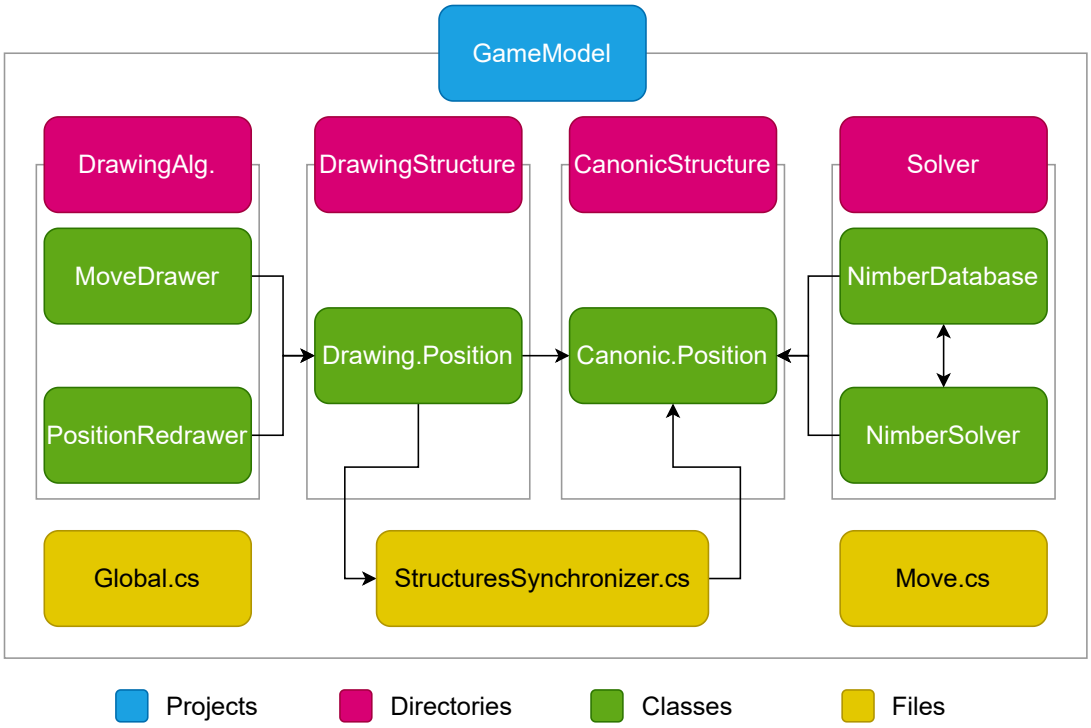


Figure 3.2: The structure of the `GameModel` project.

The class `PositionRedrawer` implementing the redrawing algorithm from Section 2.2 and the class `MoveDrawer` for the move drawing algorithm from Section 2.4 are located in the directory `DrawingAlgorithms`. These classes use the class `Drawing.Position` which is an implementation of the graphical representation that is stored together with other graphical algorithms in the `DrawingStructure` directory.

The class `Drawing.Position` contains the class `Canonic.Position` that implements the string representation and that is stored in the directory `CanonicStructure` together with the algorithms for simplifying strings and with the algorithms for generating string's children. All these parts were discussed in Section 2.3 as well as the algorithms for computing the outcome of a position and the algorithms for generating the databases. These two latter algorithms lie in a different directory named `Solver` but they still use `Canonic.Position`.

There are three other source files that are located directly in the project directory. The file `Global.cs` contains general data structures that are used in the implementation of the graphical and the string representation. The representation of a move, as described in Subsection 2.4.1, lies in the file `Move.cs`. Most of the mentioned internals use this representation, however, we do not list these dependencies in Figure 3.2 for clarity. Finally, the file `StructuresSynchronizer.cs` contains

the important synchronization mechanism from Subsection 2.4.3. The synchronizer is used by `Drawing.Position` to synchronize the encapsulated `Canonic.Position` after each insertion of an edge.

In the following five subsections, we describe the individual classes with their mutual dependencies in more detail.

3.3.1 DrawingStructure directory

The directory `DrawingStructure` primarily contains the implementation of the graphical representation. The hierarchical structure of the graphical representation components is exactly the same as described in Subsection 2.1.1.

Drawing.Position The wrapping class of the representation that points to the outer region is `Drawing.Position`. This class is an entry point to the hierarchical structure of the graphical representation that provides many service functions to make the whole structure available and modifiable from the outside without the necessity to traverse the structure. For example, it aggregates all the vertices and the edges of the graphical representation together with their sizes. It also offers a comfortable API for inserting an edge that is subsequently drawn into the representation.

Drawing.Region A region of a position is implemented inside the class `Drawing.Region`. A region R contains a list of boundaries and regions that are incident with the region R. These lists are subsequently modified as edges are being inserted into the graphical representation. Since each edge is drawn inside a single region, `Drawing.Region` is the right place for functions that insert a drawn edge into the structure.

Drawing.Boundary `Drawing.Boundary` is a relatively simple implementation of a boundary whose only purpose is to hold a sequence of edges that determine the shape of a boundary.

Drawing.IEdge, Drawing.Edge, Drawing.ReversedEdge The edges that form a boundary are implemented in the class `Drawing.Edge`. As we discussed in Subsection 2.1.1, we sometimes need to insert an edge into a boundary with the opposite orientation than its original one. This could be solved by creating a new instance of `Drawing.Edge` with an appropriate orientation and insert it into a boundary. However, this solution removes any relation between occurrences of a single edge that differ only in the orientation, and thus any modification of one edge does not affect the other one.

Therefore, to enable reversed insertions, we use a proxy class `Drawing.ReversedEdge` that behaves like an independent edge. However, the proxy class does not contain any data except of the original `Drawing.Edge` whose logic is reversed to make an illusion of a reversed edge, and thus each modification of a reversed edge takes an effect in the original one. Then, there is a common interface `IEdge` for both `Drawing.Edge` and `Drawing.ReversedEdge` so that it is possible to insert an edge into a boundary either reversely or directly as shown in Figure 3.3.

In the file `Edge.cs`, there is also a small data structure `SmallEdge` which represents two consecutive vertices in a position that are connected by a line segment.

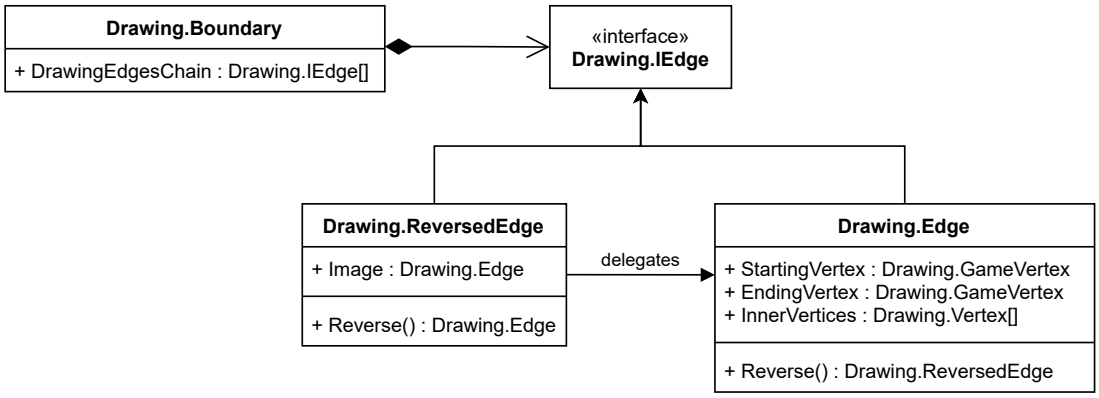


Figure 3.3: The structure of reversed and unreversed edges.

Drawing.Vertex, Drawing.GameVertex The implementation of an inner vertex is located in the class `Drawing.Vertex` that primarily contains a position of the vertex. There is also a `RedrawingInfo` class that is used for the redrawing algorithm whose implementation is discussed in Subsection 3.3.2.

Since a game vertex is only an extension of an inner vertex, its implementation lies in the `Drawing.GameVertex` class that inherits from `Drawing.Vertex`. The game vertex also contains the information about the number of its lives and whether it is touched by a cursor which is a little help for the `UserInterface` project.

RelativePosition, RelativeValue, Polygon, LineSegment The source file `DrawingGlobal.cs` contains many important structures that are used in the graphical representation. The first one of them is the `RelativePosition` structure in which all the relative coordinates of the graphical representation are expressed. It is simply a pair of two `doubles`, each in the range $[0, 1]$. Then, in `UserInterface`, there is a useful function that converts the relative positions to the absolute ones by multiplying the x- and y-coordinates by the width and the height of the window, respectively.

Similarly, there is a `RelativeValue` structure which is only a one-dimensional version of the `RelativePosition`. It is used for expressing relative and one-dimensional values like the default length or the width of a small edge. The relative values are then converted to the absolute values in `UserInterface` by multiplying them by the shorter size of the window.

The next structure `Polygon` represents a polygon formed by the border boundary of a region. Thus, there is a natural conversion from a region to `Polygon`. The structure `Polygon` contains useful algorithms like a function that checks whether a point lies inside a given polygon or a function that determines the orientation of a polygon which can be used to determine the orientation of a boundary, and thus to distinguish between inner and border boundaries.

Finally, a line segment is represented by the `LineSegment` structure that contains many useful geometric algorithms.

3.3.2 DrawingAlgorithms directory

The directory `DrawingAlgorithms` contains two important algorithms, the redrawing algorithm from Section 2.2 and the drawer of a computer move from Section 2.4.

PositionRedrawer The implementation of the redrawing algorithm is located in the class `PositionRedrawer`. The core function `Redraw()` works exactly as described in Algorithm 3. The computed force and surroundings of `Drawing.Vertex` lie directly inside the vertex in the form of the `RedrawingInfo` structure.

The quadtrees used in the redrawing algorithm are implemented inside the `QuadTree<T>` class which is a generic class that accepts the `IQTInsertable` type. Structures that implement `IQTInsertable` are `QTVertex`, `QTPoint`, and `QTEdge` which determine the elements that are inserted into the quadtree. There is almost no difference between `QTVertex` and `QTPoint` since both of them represent a point in a plane. However, we use `QuadTree<QTVertex>` for real vertices occurring in a position and `QuadTree<QTPoint>` for points that form the border of the playing area since they do not correspond to any real vertex.

`PositionTransformation` is an important class that is used to animate redrawing of a position in `UserInterface`. This class is capable of capturing positions of vertices and edges inside `Drawing.GameModel` before launching the redrawing algorithm. Then, after redrawing is performed, `PositionTransformation` can produce a simple structure describing how the vertices and the edges have changed their position.

MoveDrawer The large class `MoveDrawer` implements a complex logic that draws a computer move according to its representation `Move`. The drawing starts in the function `DrawMove()` and it then continues either with a relatively simple drawing of a double-boundary or with a much more difficult drawing of a single-boundary move. In both cases, the whole algorithm consists of many small steps that draw a path between certain occurrences of a vertex on a boundary of a polygon; no matter whether the drawn path is a part of the drawn move, or a part of a spindle, or a splitting curve.

Drawing of a path consists of three important parts. First, the polygon in which the path should be drawn is constructed. Then, possible starting and ending vertices on a boundary of a polygon are marked. The sets of the starting and the ending vertices do not have to contain a single element and they do not even have to be disjoint.

Second, the polygon is triangulated so that the resulting mesh keeps the information about the starting and ending vertices. For this step, we exploit the Triangle.NET [33] library which will use our special vertices `MeshVertex` containing useful information, for example whether the vertex is starting, ending, both, or none of them.

Third, we need to find the shortest path between a starting and an ending vertex in the mesh using the Breadth-First-Search; note that these two vertices must differ. This step is implemented in the function `FindPathInMesh` that is reusable for drawing any type of paths. Besides the mesh, this function also receives two functions that check whether the correct occurrences of a starting

and ending vertex are being connected.

The most complex part of the drawer, however, does not lie in these three steps. The most difficult is to correctly put together a large number of them to draw the whole move. This problem involves a large number of special cases together with nontrivial updates of polygons after each path is drawn. See the attached source code for more details.

3.3.3 CanonicStructure directory

The directory `CanonicStructure` contains the implementation of the string representation and the algorithms that simplify or modify strings as described in Section 2.3. Since the computer opponent is based on the solver GLOP [6], our implementation is also inspired by the source codes of GLOP. However, there are many differences mainly because we need to include mechanisms for a possible synchronization with the graphical representation. Other differences follow from a different choice of programming languages since GLOP is implemented in C++.

VerticesSequence The abstract class `VerticesSequence` represents elements of the string representation that can be perceived as a sequence of vertices. Note that all the components of the string representation satisfy this condition, and thus they inherit from this class. The only exception is `Canonic.Vertex` because we do not iterate over sequences consisting of a single vertex, and thus `Canonic.Vertex` does not inherit from `VerticesSequence`. The class `VerticesSequence` declares functions like an iteration over the sequence of vertices or a conversion to `string`. There is also a definition of a comparison of two sequences that is based on the comparison of two vertices from `Canonic.Vertex`.

Here, we point out a difference between GLOP and our program. Besides the hierarchical structure, GLOP also contains a flat representation of a position using a sequence of vertices that are stored in an array. However, we do not keep these two representations side by side since a synchronization between them needs to be performed often. On the other hand, we still need to have a view of the structure via the flat sequence of vertices. For this purpose, the iterator of vertices, which each descendant of `VerticesSequence` implements, serves very well.

Structure Note that many components of the string representation have a similar recursive structure. Position is a list of lands, lands are lists of regions and so on. They also implement many functions in a similar way. For example, building a string often only consists of a recursive call on lower layers and of adding an extra separator. The implementation of an iteration inside `VerticesSequence` is analogous.

Therefore, we have another abstract and generic class `Structure<T>` which is a component of the string representation that contains `List<T>` where `T` is a descendant `VerticesSequence`. Thus, `Structure<T>` is capable of implementing the two previous functions. Note that `Structure<T>` has properties of `VerticesSequence` as it is a descendant of `VerticesSequence`. Finally, all the components of the string representation except of `Canonic.Boundary` and `Canonic.Vertex` inherit from `Structure<T>` where `T` is the corresponding substructure.

Canonic.Position The class `Canonic.Position` implements the position component of the string representation. Besides its hierarchical properties, it is an entry point to the string representation similarly as `Drawing.Position` to the graphical representation. It implements many service functions, where the most important one is the `ComputeChildren()` function. This function returns the list of children of a position which is a key property for the analysis of Sprouts positions.

Canonic.Land The class `Canonic.Land` is another component of the hierarchical structure of the string representation. This class contains many algorithms such as the algorithm that splits a list of regions to independent lands or functions that coordinate the reduction and canonization of a string.

Canonic.Region The class `Canonic.Region` is the implementation of regions of the string representation. There are many functions that are used during the reduction and canonization of a string. `Canonic.Region` also contains a function for generating partitions of boundaries that is used when single-boundary children are computed.

Canonic.Boundary The class `Canonic.Boundary` is the implementation of the first component of the string representation that does not inherit from `Structure<T>` but it is still a descendant of `VerticesSequence`. Similarly as `Canonic.Region`, it contains further functions of the reduction and the canonization. `Canonic.Boundary` could also contain a simple list of vertices if it inherited from `Structure<T>`. Instead, it contains a special data structure `RotatableList<U>` that is optimized for the rotations and reversions that are often performed during the canonization on the boundaries.

Canonic.Vertex The structure `Canonic.Vertex` is the implementation of the final component of the string representation. The type of a vertex is determined by appropriately selected integer constants. Note that each character that occurs in the string representation, including the separators, is represented as `Canonic.Vertex` even though they do not have to correspond to any real vertex. In `Canonic.Vertex`, there are further functions of the canonization that use the representation of a vertex.

Generating children To determine the best possible move of a computer opponent in a position P, the string representation must be capable of generating children so that `Solver` can explore the game tree of P. Furthermore, after the best move is found, the move must be drawn in the corresponding graphical representation $gr(P)$.

For this purpose, there is an option to add references to the graphical representation into the string representation. To pair `Canonic.Region` with a corresponding `Drawing.Region`, we replace `Canonic.Region` by `Canonic.ConnectedRegion` that only adds a reference to `Drawing.Region`. Since `Canonic.Vertex` is a structure, there does not exist any `Canonic.ConnectedVertex`. Instead, the reference to an occurrence of `Drawing.Vertex` can be placed directly in the field

`VertexIndex` of `Canonic.Vertex`. Note that none of these references is used for children lying deeper than in the first layer of a game tree; the references are missing or they are not filled in this case. Recall that the theory behind these references was described in Subsection 2.4.1.

The logic that generates children of `Canonic.Positon` is scattered around each component of the string representation except of the `Canonic.Vertex` component. Then, every component computes its children using the children generated in an underlying component. For example, children of lands are computed from children of regions. The children are represented by classes lying in `Children.cs`. These classes contain their connected versions similarly as for `Canonic.Region`. They are used to extend children by the move representation `Move` that generates them in the graphical representation. This is then used for drawing the best move that was found by `Solver`.

3.3.4 Solver directory

The directory `Solver` contains an implementation of the algorithm for computing the outcome of a position and it also includes its parallel version. Both algorithms, were described in Subsections 2.3.4 and 2.3.5, respectively. There is also the training algorithm from Subsection 2.3.6.

NimberSolver The class `NimberSolver` is the implementation of a solver using the nimber theory. It contains the sequential version and also the parallel one. A chosen version is determined by the configuration of the solver. The main function of the solver `GetBestChildren()` returns the list of structures `BestMove` which describe a chosen child together with the information whether it is a winning or losing child. In the case of losing moves, there is also an integer value that expresses how good the move is according to the probability p from Subsection 2.3.6.

Note that for the first generation of children, Algorithm 8 from Subsection 2.3.4 is not used since we need to expand each child separately to be able to determine the losing ones. However, Algorithm 8 is performed standardly in next layers together with the implementation of a couple in the `Couple` structure. The parallel version of Algorithm 8 in addition uses synchronization primitives such as locks or the `OpenedCouples` databases that were also described in Subsection 2.3.5.

The solver can be configured by passing the `SolverConfiguration` structure that specifies the properties such as the constants k_W and k_L or whether to use the probability p from Subsection 2.3.6. For non-perfectly playing opponents, there is also an option to set a maximum computation time or a maximum depth in which the game tree of a position is explored. It also contains an option whether the computed nimbers should be stored. This option is set to `true` when a database is being trained and to `false` during a game for all the computer opponents. Finally, we can set a number of `Task` classes that will be used during the parallel exploration, however, the parallelism cannot be used in our application as discussed in Subsection 2.3.5.

NimberDatabase A database of nimbers is implemented in the `NimberDatabase` class. The main field of this class is `Dictionary<string, int>` that stores converted instances of `Canonic.Position` into a string together with their nimber.

Besides a simple API that can read nimbers from a database or store them into a database. There are also functions that can load `NimberDatabase` from a file or store it into a file. Stored databases are simple text files with a filename extension `.spr`. Each line of the files contains a pair of a position string and its nimber.

`NimberDatabase` also contains functions that can fully train a position using `NimberSolver` with a certain `SolverConfiguration`. These trainings were discussed in Subsection 2.3.6. There is also an option to create a *full tree database* of a position P which means that each position that occurs in the game tree of P is trained. This is a useful option because we are able to generate a full tree database of the n-spot positions with $n \leq 6$, and thus to easily train all the positions that can be derived from them.

Levels of the computer opponents In our application, there are five different levels of AI that only differ in `SolverConfiguration` that is passed to `NimberSolver`. In the quick game mode, there are *Easy*, *Medium*, *Hard*, and *Insane* AI levels. The Insane AI is replaced by the *Perfect AI* in the campaign mode.

The configurations of AI levels are stored in `SolverConfiguration`. The Easy, Medium and Hard AI levels have unlimited values of the constants k_W, k_L. The Easy and Medium levels do not use the probabilities p but the Hard AI does. The maximum exploration depth of all of the three levels is restricted to three and the maximum computation time is set to one second. Therefore, these levels are not able to compute too much during a run-time and they must rely on the databases. This gives us an option how to configure their difficulty relatively well.

The Easy level does not use any database and thus it plays well only on very small positions. The Medium level uses the full tree database of `0*3` that is stored in the `data\medium_database.spr` file. Therefore, the Medium plays most likely perfectly on all positions that can be derived from `0*3`. The Hard AI uses a full tree database of `0*5` that is stored in the `data\hard_database.spr` file.

The most advanced level, the Perfect AI, is present only in the campaign mode where the positions are prepared in advance. The goal of the Perfect AI is to play perfectly on these positions at any cost. Therefore, it has no restrictions on the maximum depth and the maximum computation time. We have already discussed its values of k_W, k_L and the use of the probability p that depend on the size of a played position in Table 2.6 from Subsection 2.3.6. To make the Perfect AI play perfectly so that it never gets stuck in thinking, we fully train each position in the campaign with appropriate configurations. This produces a large database used by the Perfect AI during a game. The database is stored in the `data\perfectai_database.spr` file and it contains 1 329 386 nimbers.

Since a position that was not trained in the largest database can occur in the quick play mode, we cannot use the same configuration for the Insane AI as for the Perfect AI. Therefore, the Insane AI slightly differs from the Perfect AI in the configuration but not in the used database. The configuration is almost the same as the configuration of the Hard AI but it does not have a limited maximum depth. The maximum computation time is increased to three seconds to give the Insane AI a higher chance to compute simple positions that were not trained in the database.

3.3.5 Synchronization of the representations

We have already discussed how to determine a move represented by `Move` that creates a child of `Canonic.Position`. Now, we describe `Move` in more detail. `Move` is an abstract class that represents either a double-boundary or a single-boundary move and thus it has two descendants `DoubleBoundaryMove` and `SingleBoundaryMove`. Both of them inherit the information about the connected vertex occurrences in the form of `VertexIndex` and also about `Drawing.Region` in which the move is made. Moreover, `SingleBoundaryMove` contains the major and minor partition that arise after a single-boundary move is made. `SingleBoundaryMove` also implements an important function `GetEnfoldedBoundaries()` that returns the set of boundaries which should be enfolded during drawing a single-boundary move in `MoveDrawer`. It also returns whether the reversed or unreversed enfolding should be used as we discussed in Subsection 2.4.2.

In Subsection 2.4.3, we decided to have `Drawing.Position` and `Canonic.Position` in parallel where `Canonic.Position` is synchronized after each edge insertion into `Drawing.Position`. The synchronization is implemented inside `StructuresSynchronizer` and, as we explained before, the implementation is relatively complicated. Now, we point out some difficulties that can occur during the synchronization.

Consider a graphical representation $gr(P)$ and a string representation $sr(P)$ of a position P. After an edge e is inserted into $gr(P)$, two steps of the synchronization follow. First, a child $sr(P')$ of $sr(P)$ that was created by a move that corresponds to the drawn edge e is found. The second step is to update all the references in $sr(P')$ into $gr(P')$. This is because there are newly added vertices and possibly new regions. Moreover, the structure of connected boundaries together with all the indices of vertex occurrences changes significantly. The synchronization is even more complicated by simplifications of a string that can occur in $sr(P)$, such as the merged boundaries or vertices.

Even the first step is nontrivial. Sometimes, it can happen that multiple children of $sr(P)$ correspond to the inserted move; or even worse, none of them. The first problem arises when a loop is made on a singleton because there are two children of $sr(P)$ that only differ in marking of the major and minor partitions. This can cause problems when one of the children is trained and the other is not and we choose the wrong one. To prevent this, we reduce these symmetrical children to a single one when children of $sr(P)$ are generated. Then, if the removed child is drawn, we need to ensure that the symmetric child is chosen during the first step of the synchronization.

The problem that no child of $sr(P)$ was paired with a move can be caused when a connected vertex occurrence or the region surrounding the drawn edge e does not occur in the corresponding child because it was simply deleted. The first case of a missing vertex occurrence is easier to see. It happens when two occurrences of $sr(P)$ were merged into a single occurrence `2` that contains only a single reference to $gr(P)$. Then, the problem arises if e connects the occurrence of $gr(P)$ whose reference was removed from $sr(P)$.

The second issue is more complicated. Consider a position `2a2a` in Figure 3.4. During the simplification, the string is reduced to `222` which is a correct simplification performed by merge boundaries. After the move in Figure 3.4 is drawn, the string changes to `2A|A` which is simplified to `22`. Now, see that there are two alive

regions even though, there is only one in 22. This is again caused by a merging simplification.

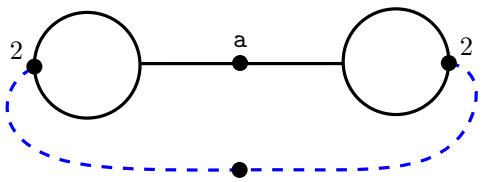


Figure 3.4: A position P after inserting the dashed move whose $gr(P)$ contains two alive regions whereas the synchronized $sr(P)$ contains only a single one.

See the attached source code for a more detailed understanding of the synchronization.

3.4 NetworkClasses project

`NetworkClasses` is a simple project containing data that are shared between `UserInterface` and `Server` during the network communication in a remote game. It also contains a general function that can serialize .NET objects by using the Newtonsoft.Json library [35].

We shortly describe all the components of the project in Subsection 3.4.1 but we primarily focus on the communication protocol in Subsection 3.4.2.

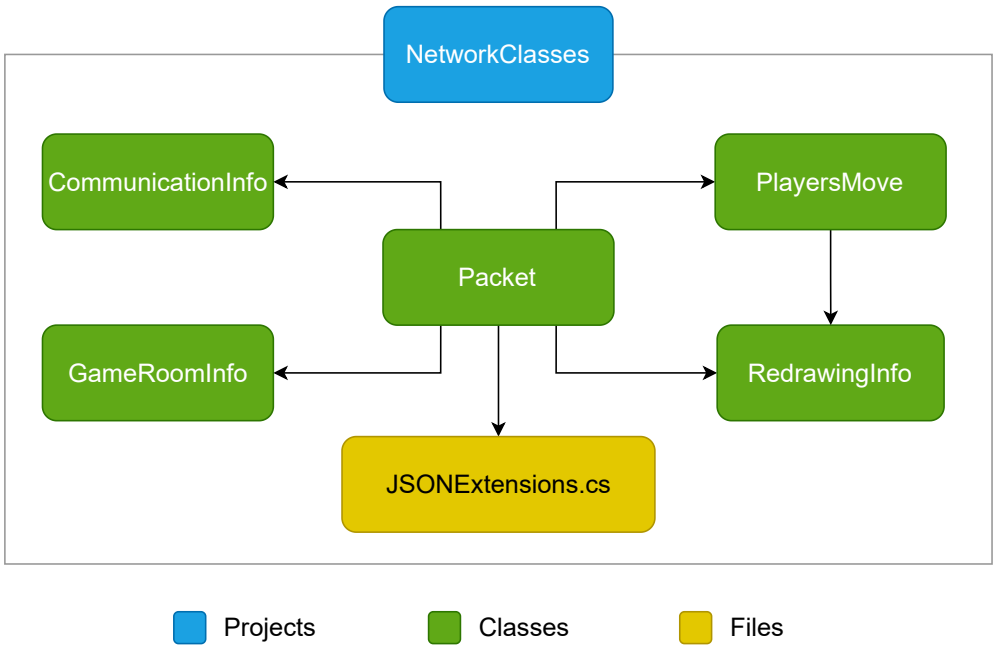


Figure 3.5: The structure of the `NetworksClasses` project.

3.4.1 Project components

Now, we describe all the classes and structures that are located in `NetworkClasses`. The components and their mutual dependencies are shown in Figure 3.5

CommunicationInfo The small static class `CommunicationInfo` contains default constants for the network communication. There is a specified port of the communication as well as a newline character that is used for separating individual `string` messages. There are also defined default settings of the JSON serialization and deserialization that are used for each .NET object in the whole application.

JSON Extensions In the `JSONExtensions.cs` file, there is a generic extension function `ToJson<T>()` that adds each .NET object `T` a property so that it can be serialized using the Newtonsoft.Json [35] library with the default settings from `CommunicationInfo`. There is also an analogous function `FromJson<T>()` that extends `string` so that it can be potentially deserialized to an arbitrary .NET object `T`.

Packet The structure `Packet` carries a single logical message that is sent to the opposite endpoint through a network. Each `Packet` is serialized to `string` using `JSONExtensions.cs` and sent as a byte stream separated by the newline character from `CommunicationInfo` via the *TCP protocol*.

To distinguish types of packets, `Packet` contains a field `PacketID`. There is also a `string` field `Message` that is either empty or filled by the serialized `GameRoomInfo`, `PlayersMove`, or `RedrawingInfo` depending on the semantics of the packet's type. All the packet types together with their semantics are described in the following Subsection 3.4.2.

GameRoomInfo In the remote game, one player creates a game room and the second player connects to it. The class `GameRoomInfo` contains the properties of such a game room. In the current version of the application, it is only a name and a position on which the players will play.

RedrawingInfo During a remote game, any player has an option to redraw a position which also results in a redrawing on the opponents side to keep both drawings the same. The class `RedrawingInfo` contains such data that are sent to the opponent so that he can redraw the position. It is not enough to only send a request for redrawing since the redrawing time is limited, and thus both redrawing algorithms could have different results. Therefore, the resulting instance of `Drawing.Position` is sent inside `RedrawingInfo` after redrawing and `PositionTransformation` that allows the opponent to animate the redrawing.

PlayersMove After a player makes a move, it is sent to the opponent using `PlayersMove`. Similarly as `RedrawingInfo`, it contains the resulting position and data that are needed to animate the drawing of the opponent's move. Since the redrawing algorithm is performed after each move, the resulting position is sent via the `RedrawingInfo` structure.

3.4.2 Communication protocol

In this subsection, we describe the communication protocol between `Server` and a player that is represented by the `UserInterface` project. First, we describe how a client connects to the server. Then, we discuss all the possible communications in

the menu with game rooms and also the communication during a game. Finally, we describe disconnects and how they are resolved. Note that all the communications are always led through the server and never directly between clients.

Connecting to the server A connection to the server is very simple. After a client successfully connects to the server, the server answers with a `Packet.ID.Connected` packet together with `Packet.ID.GameRoomsList` that contains a list of `GameRoomInfo`; see Figure 3.6.

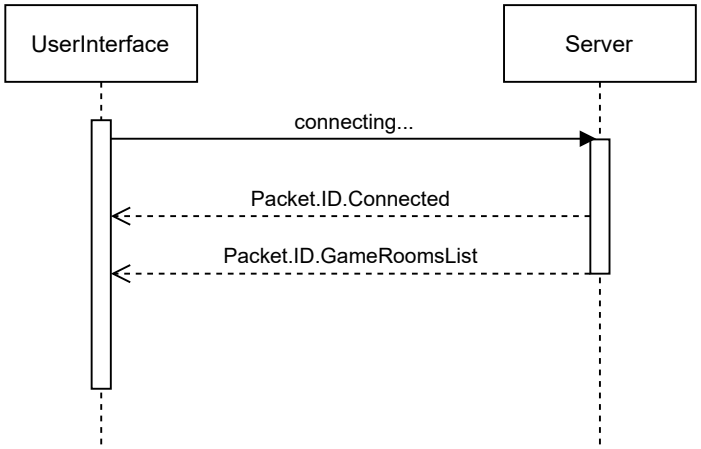


Figure 3.6: The communication protocol of a successful connection to the server.

Communication in the menu The menu with game rooms is used to create or join a room, and thus to interconnect players to start a game. A client can refresh the list of game rooms by sending `Packet.ID.RefreshRooms` on which the server responds with a `Packet.ID.GameRoomsList` packet that contains a list of rooms.

For creating a new room, a client sends `Packet.ID.CreateRoom` request that contains corresponding `GameRoomInfo`. If the creation succeeds, the server answers with `Packet.ID.RoomCreationSucceeded`. Otherwise, for example if the name of the room already exists, the server sends `Packet.ID.RoomCreationFailed`.

Then, the client is waiting until an opponent connects. If he wants to stop waiting and cancel the room, he sends `Packet.ID.CancelRoom` request. Again, he must wait until the server responds. In the positive case, the server sends `Packet.ID.RoomCancelationSucceeded`; otherwise, it answers by sending a `Packet.ID.RoomCancelationFailed` packet, which happens if an opponent had connected to the room just before the client tried to cancel it.

To join a room, a client sends a `Packet.ID.JoinRoom` request that contains `GameRoomInfo` about the connected room. Then, if the joining succeeds, the server sends a `Packet.ID.JoinSucceeded` packet to the client and it also sends `Packet.ID.OpponentJoinedRoom` to the host of the room. Clients represented by `UserInterface` then react by starting a game. Joining a room can also fail if, for example, the joint room does not exist. The server then sends `Packet.ID.JoinFailed`.

An example of a commutation in the remote game menu is shown in Figure 3.7.

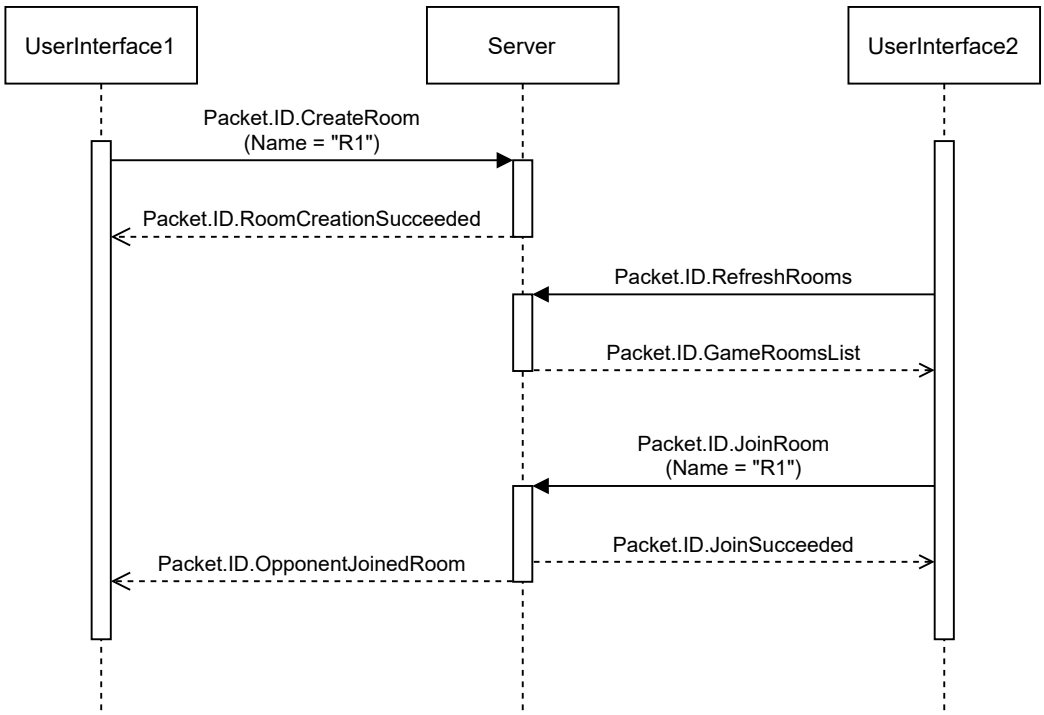


Figure 3.7: An example of a communication between the server and two clients that interconnect and start a game.

Communication during a game In our application, a Sprouts game starts with a *Swap2 opening* [37] which is a standard opening used in *Gomoku*. The connected opponent can either choose the starting player by sending `Packet.ID.YouStart` or `Packet.ID.OpponentStarts` to the server; note that the semantics behind these packets is from the point of view of the opponent. This packet is then passed to the opponent and the game starts. However, the connected opponent also has another option by making a move, which is send via a `Packet.ID.PlayersMove` containing `PlayersMove`, and thus to make the opponent choose the starting player on the resulting position by using the same types of packets. Now, the opponent does not have the option to postpone the decision.

After a game is opened, players start making moves and sending them to the opponent by `Packet.ID.PlayersMove`. The player on the move also has an option to redraw a position by sending the redrawing information in the `RedrawPosition` structure containing `RedrawingInfo` to the server. The server then passes the packet to the opponent.

The players continue in this manner until the game is over or until one of the players gives up by sending `Packet.ID.GiveUp`. Then, they have an option to play again on the same position instead of returning to the main menu. A player expresses the intention to play again by sending `Packet.ID.PlayAgain`. If both players decide to play again, the game server notifies both the players by sending the packet `Packet.ID.PlayAgain`. Then, the game starts from the same position. If a player decides not to play again, he sends a `Packet.ID.NotPlayAgain` packet. The server acknowledges it with the same packet and also passes the packet to the opponent. Note that the server can send `Packet.ID.PlayAgain` acknowledgement instead if the player previously wanted to play again and his opponents had sent `Packet.ID.PlayAgain` before the player sent the `Packet.ID.NotPlayAgain`

packet.

An example of such a communication is shown in Figure 3.8.

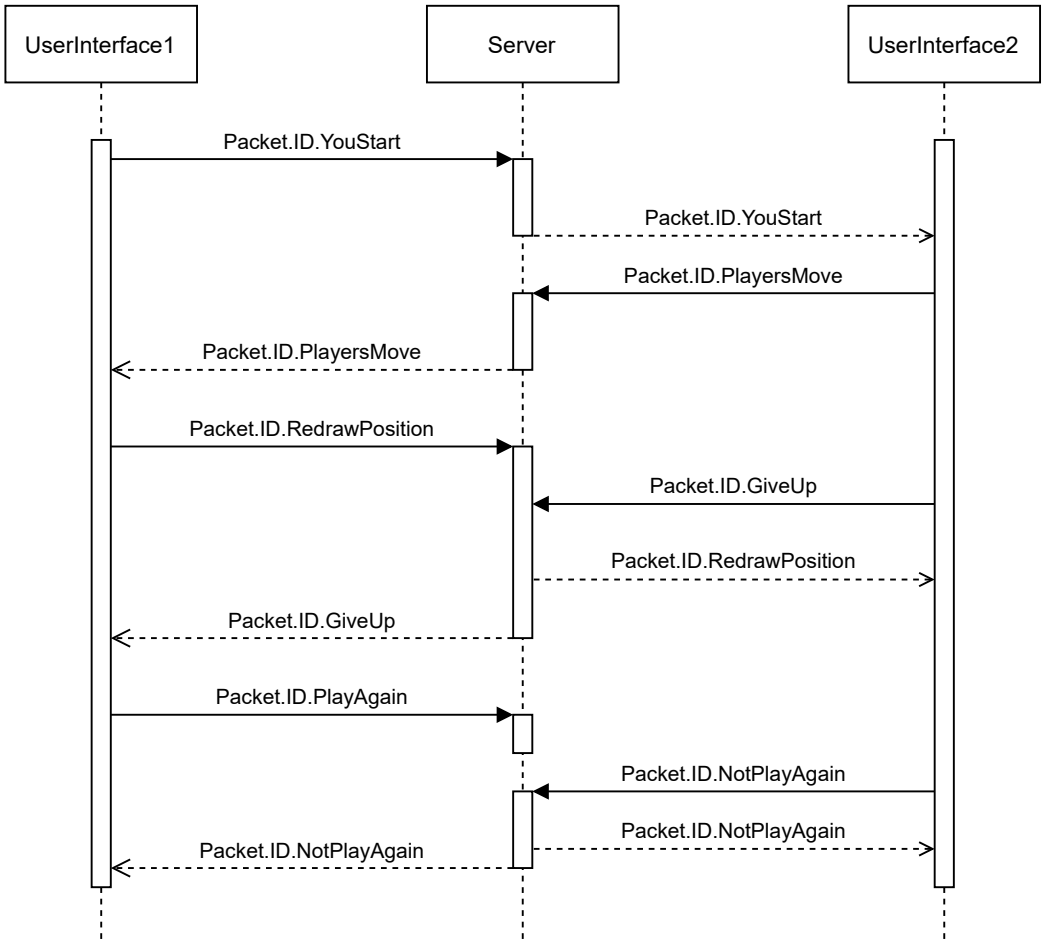


Figure 3.8: An example of a more complicated communication during a game.

Resolving disconnects If a player or the server decides to end the connection, they can do it gracefully by sending `Packet.ID.Disconnecting`. However, this option is not used in the current version of the application since there are no resources that should be released immediately and its implementation is unnecessarily complicated. Instead, the endpoints simply wait until the underlying network layer finds out a disconnect or they determine a disturbed connection using *heartbeat packets*.

In a healthy connection, a client regularly sends `Packet.ID.Heartbeat` to the server and the server responds in the same way. Then, if no packet is received for a certain period of time, one endpoint decides that it is a disconnect. The advantage of this approach is that it can discover a large spectrum of disconnects that could not be resolved by a sending a graceful packet or by waiting to the information from the underlying layer. The heartbeats are not shown in the example communication diagrams for simplicity.

When a player disconnects during a game, his opponent needs to be notified by the server. This is achieved by sending a `Packet.ID.OpponentDisconnected` packet.

3.5 Server project

The console application `Server` is a mediator between players during a remote game. Since the communication protocol was described in Subsection 3.4.2, we only discuss the classes of the server in Subsection 3.5.1 and the architecture of the server in Subsection 3.5.2.

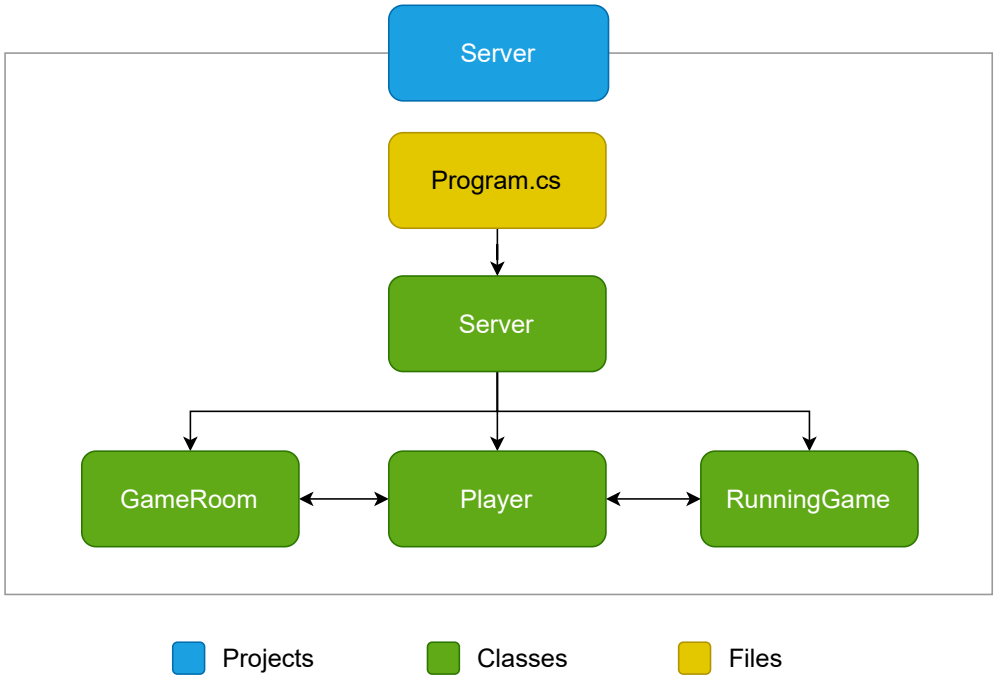


Figure 3.9: The structure of the `Server` project.

3.5.1 Project components

Now, we describe all the classes that are located in the `Server` project. The classes and their mutual dependencies are shown in Figure 3.9.

Player The class `Player` represents a player that is connected to the server. There is important information such as `GameRoom` hosted by the player or a reference to `RunningGame` that the player currently plays. This class also contains logic that can send `Packet` to a player or receive it from a player. Note that the receiving function `ReceivePacketAsync()` is asynchronous which is later used in `Server`.

GameRoom A created game room is represented by a simple class `GameRoom`. Besides `GameRoomInfo`, this class also contains `Player` that hosts the room.

RunningGame After a player connects to `GameRoom`, it is transformed into a `RunningGame` instance and a game starts. The class `RunningGame` does not only hold information about a running game but it is mainly used to pair interconnected players.

Server The class `Server` is the core of the project that contains the main loop handling all the communication with clients.

Entry of the application The `Main` function of the application is located in `Program.cs`. We only start the server here and set up the interrupt handler that shuts down the server.

3.5.2 Architecture of the server

The main loop of the server is located in the `Run` function of the class `Server`. It consists of three parts: establishing a connection, communicating with connected clients, and closing terminated connections. To get a better performance, we use an asynchronous model of the server for these steps.

The major thread only repeats an infinite loop that terminates after the server is shut down. At the beginning of the loop, if there is a pending connection, a new `Task` created by the asynchronous function `HandleNewConnectionAsync` is run to process the connection. In the second phase of the loop, the sever checks whether there is any `Task` from the previous step that successfully connected a client. If there is, another `Task` is created by an asynchronous function `ListenAsync` that will handle a communication with the client. Finally, at the end of the loop, there is a handler that closes finished connections and also a function that disconnects inactive clients that have not sent any `Packet` for a long time. Since each client must keep sending the heartbeat packet regularly, such a connection between the server and the client was probably disconnected without noticing.

Let us describe the implementation of the function `ListenAsync` in more detail. The function is based on a loop that receives a packet from a client using the `ReceivePacketAsync` function that is then processed in the function `ProcessReceivedPacket`. To improve the performance, this function uses the fact that `ReceivePacketAsync` is asynchronous, and thus it can start receiving the next packet even if the previous one has not been processed yet. However, it can never happen that it will start processing a packet before processing of the previous packet was finished.

The `ProcessReceivedPacket` function that processes packets is only large switch that implements the communication protocol as described in Subsection 3.4.2. Note that the current version of the server expects that the clients correctly communicate via the defined protocol and that they send correct data. Therefore, it would be convenient to extend the server by some security checks that would prevent cheating or improve the endurance of the server.

Since the server is full of the parallelism, some kinds of synchronization must be included. For this purpose, we use many locks that prevent any race-condition. The locks are used for shared structures such as the list of game rooms, players and running games. The most complicated is to exclude common communications in `ListenAsync` with a disconnection of a client since when a client is disconnected, we often need an access to many shared data structures that need to be protected. Here, we could lock each shared structure separately but we would have to ensure a fixed locking order to prevent deadlocks which is hard in this context. Instead, we create a special lock for each communicating `Task` that locks `ListenAsync`. Then, when a client is disconnected, we simply lock all these locks to mutually

exclude the disconnection with any other communication. A trade-off of this solution is that the disconnections block other communications even in the cases where it would be unnecessary. However, this is not such a big problem since clients do not disconnect very often; still, it is a place for a possible improvement.

3.6 UserInterface project

The `UserInterface` project is a WPF application producing the resulting executable file `Sprouts - A Drawing Game.exe`. The project includes XAML documents designing the appearance of the user interface, controllers defining the behavior of the whole application, and functions that process user drawings or functions that display the graphical representation from the `GameModel` project on a canvas. `UserInterface` also represents one of the endpoints of the remote game communication, and thus there is a logic that communicates with the server. The `UserInterface` project uses absolute coordinates to draw graphics unlike the `GameModel` that uses relative coordinates. Therefore, there are simple functions that convert between these two systems in the file `Conversions.cs`.

Now, we describe the structure of the project that is shown in Figure 3.10. In Subsection 3.6.1, we go through the auxiliary classes that are used by the graphical controllers and then we describe the standalone controllers in Subsection 3.6.2.

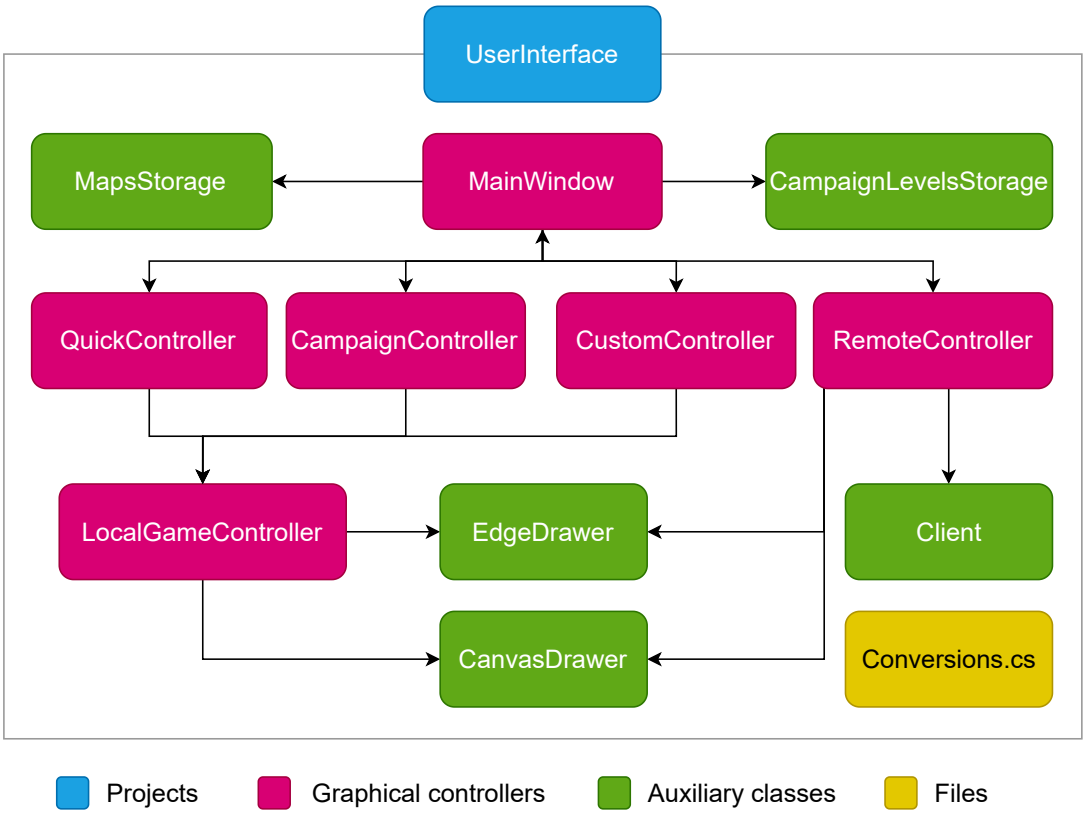


Figure 3.10: The structure of the `UserInterface` project.

3.6.1 Auxiliary classes

Before describing the graphical controllers, we discuss important auxiliary classes that they use. These classes can be split in three different categories. First, the classes `MapsStorage` and `CampaignLevelsStorage` are used to keep shared data across independent launches of the application. The classes `EdgeDrawer` and `CanvasDrawer` implement logic processing or making drawings. Finally, the class `Client` is used for a communication with the server during the remote game mode.

MapsStorage The `MapsStorage` class keeps maps which are positions that are drawn in the custom maps tab of the application and then used for playing on custom positions in the quick play or the remote game mode. `MapsStorage` uses the `maps.conf` file that is located in the special applications data folder. Maps are loaded when the application is launched and stored after the application is closed. For serializing and deserializing data, `MapsStorage` uses `JSONExtensions.cs` from the `NetworkClasses` project.

CampaignLevelsStorage The class `CampaignLevelsStorage` works similarly as the `MapsStorage` class. `CampaignLevelsStorage` keeps positions in the campaign mode together with their state such as the strongest unlocked AI difficulty or the configuration of the perfectly playing opponent on that level. The default state of `CampaignLevelsStorage` is stored in `data\levels.conf` whereas the user's progress in the campaign mode is stored in the `levels.conf` file of the special applications data folder.

EdgeDrawer The user drawings are processed in the class `EdgeDrawer`. It contains an API that can draw an edge into a given `Drawing.Position`. The function `MoveTo()` obtains a current position of the cursor according to which a next part of the constructed edge is drawn. The drawer includes logic that prevents crossings and keeps other properties of drawings such the default length of drawn small edges or the minimum angle between them.

CanvasDrawer The class `CanvasDrawer` is the second of the drawing auxiliary classes. This class obtains `Drawing.Position` and `Canvas` on which the position should be drawn. Besides the functions for drawing a position, there are also functions that animate redrawings of a position or drawings of a move. Finally, `CanvasDrawer` contains implementation of *Bézier splines* that are used for drawings of a position to make edges smoother.

Client Since `UserInterface` is the client endpoint during the remote game mode, it contains the `Client` class that allows a general communication through a network. As `Client` implements a general communication, there is no logic of the communication protocol from Subsection 3.4.2. This class only obtains an IP address and a port to which it connects via the *TCP protocol*. Then, it simply forwards all the received packets to registered listeners. For sending packets to the opposite endpoint of the communication, there is the function `SendPacket()`.

The architecture of the client is based on three parallel threads. The first of them monitors disconnects using the heartbeat packets. The second one listens to the connection and forwards received packets to the registered listeners. The final thread processes requests for sending packets and it works on a simple principle of the *Producer–Consumer pattern*.

Besides the listeners of received packets, there are also listeners of general events such as connections or disconnections.

3.6.2 Graphical controllers

The user interface of our application consists of four tabs: the quick game mode, campaign mode, remote game mode, and a tab for drawing custom maps. Each tab has a `XAML` file defining its look and the corresponding controller that implements its behaviour. Besides these four `XAML` files, there are also simple `HelpDialog.xaml` and `AboutDialog.xaml` files that define the look of dialogs that are shown after clicking on the `Help` and `About` button, respectively.

Now, we describe the main four controllers together with the `MainController` file that defines the common menu.

MainWindow The class `MainWindow` together with the corresponding `XAML` file defines the appearance and the behaviour of the side menu together with the text boxes and the option menu in the title. The main purpose of the class is to switch between tabs inside and to display information in the title. `MainWindow` also contains `MapsStorage` and `CampaignLevelsStorage` that are shared with underlying controllers.

LocalGameController The local game controllers `QuickController`, `CampaignController`, and `CustomController` have a very similar behaviour during a game or during drawing of custom maps. Therefore, this common behaviour is implemented in the general class `LocalGameController`. There are functions using the classes `EdgeDrawer` and `CanvasDrawer` to update and display the underlying `Drawing.Position` according to the user events. It also contains `NimberSolver` and `MoveDrawer` to make moves of the computer opponent.

QuickController The class `QuickController` and the corresponding `XAML` file implement the properties of the quick game mode tab. Since it uses `LocalGameController`, it only defines the behavior of the quick game menu and communication with `LocalGameController`. The controller uses `MapsStorage` from `MainWindow` as a custom map can be chosen in the quick game mode.

CampaignController The campaign mode is implemented in the class `CampaignController` and its `XAML` file. Similarly as `QuickController`, `CampaignController` uses `LocalGameController`, and thus it only implements communication with it and the logic of the campaign menu. The controller uses `CampaignLevelsStorage` that lies in `MainWindow`.

CustomController The tab that allows to draw custom maps is implemented in `CustomController` and in the corresponding `XAML` file. Even though `CustomController` does not implement any type of game, it also uses `LocalGameController` to implement the behaviour of the maps drawing. Therefore, it only defines the behaviour of the menu and the communication with `LocalGameController`. Since the controller cooperates with the stored positions, it uses `MapsStorage` from `MainWindow`.

RemoteController The remote game mode is implemented in `RemoteController` and its `XAML` file. The controller does not use `LocalGameController` because the logic of a remote game is significantly different from the logic of a local game. Therefore, it is easier to implement the logic of a remote game directly inside `RemoteController` rather than making `LocalGameController` even more general. This makes the remote controller the most complex controller since it has to implement the behaviour of many menus together with the implementation of a game. For example, the controller must use the classes `EdgeDrawer` and `CanvasDrawer` directly. Furthermore, the controller contains the implementation of the communication protocol from Subsection 3.4.2 since it has to communicate with `Server` via the `Client` class. Similarly as the `QuickController`, `RemoteController` also uses the `MapsStorage` class from `MainWindow`.

4. User guide

In this chapter, we describe the installation of our application *Sprouts: A Drawing Game* and include a tutorial how to use it. Then we describe the *Server* application that is used for the remote game mode.

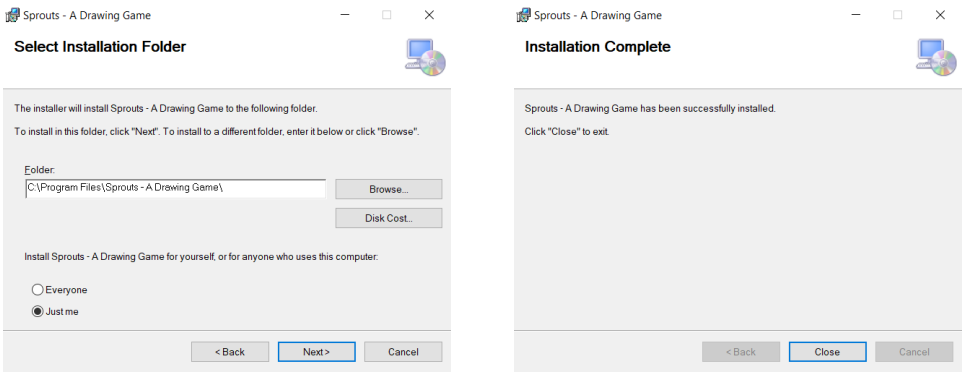
4.1 Sprouts: A Drawing Game

First, we go through the installation of the *Sprouts: A Drawing Game* application and then we describe menus of the application together with the individual game modes.

4.1.1 Installation

The installation file `setup.exe` is located in the `installation\Sprouts` directory in Attachment A. It is a 64-bit Windows installer that can be launched with a simple double-click.

The application is running on the .NET Core 3.1 framework, and thus if any required component is missing the installer requests its installation just before the installation of the application itself. Then, the installer asks for the installation folder and users for whom to install the game; see Figure 4.1. Before confirming the installation, the user will probably need to allow the installer to make changes in his device. Finally, after the installation is successfully completed, the user will see the window from Figure 4.1 and also the shortcut file in his `Desktop` folder.



Selecting the installation folder. A successful installation.

Figure 4.1: The installation of Sprouts: A Drawing game using `setup.exe`.

4.1.2 Tabs and Options menu

The application consists of four tabs that can be switched using a side menu that can be collapsed with the button ①; see Figure 4.2. The button ② displays the quick play mode, the button ③ the campaign mode, the button ④ the remote game mode, and finally, the button ⑤ enters the tab in which custom maps can

be drawn. The text block ⑥ contains the title of the currently opened tab. The tabs can be switched at any time except of the situation when a game is running.

In the top right corner of the application, there is an option menu with two items. The menu item ⑦ opens a dialog explaining the rules of Sprouts as described at the beginning of Chapter 1 and also the basic game strategy from Subsection 1.2.3. The menu item ⑧ shows a dialog with short info about the application.

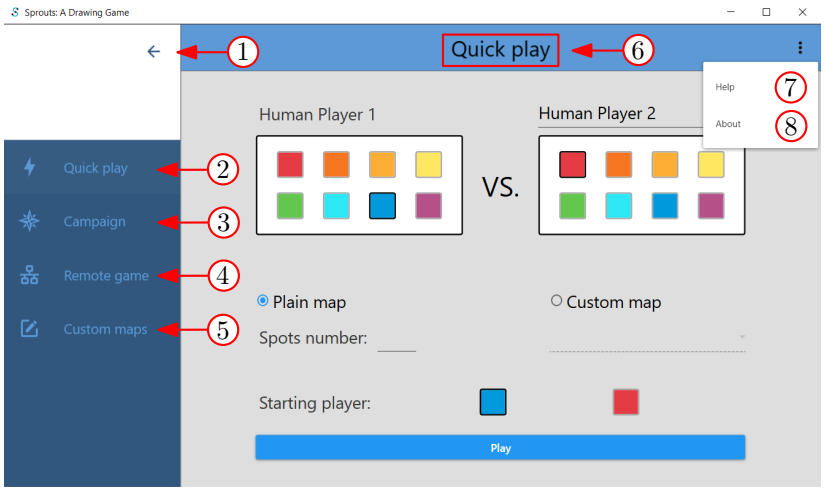


Figure 4.2: The tabs and the option menu.

4.1.3 How to play

Each of the individual game modes starts a Sprouts game with different settings. Now, we describe a single game and how to play it in general which is common to all the game modes.

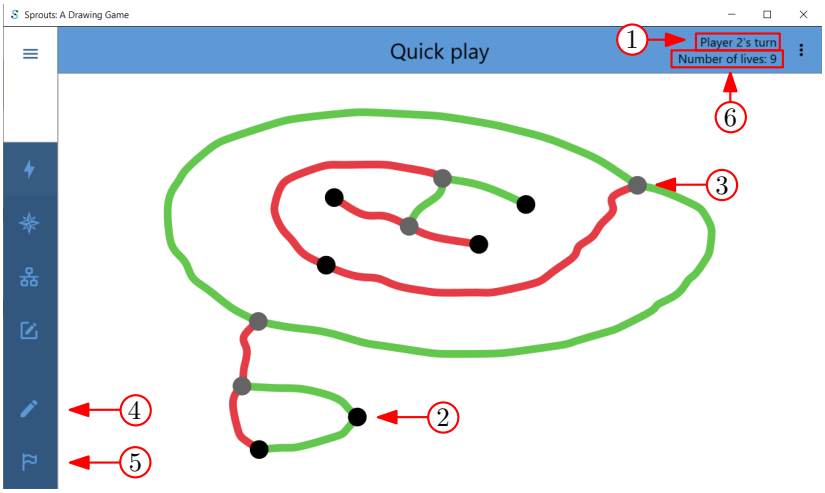


Figure 4.3: An example of a Sprouts game.

A game consists of a Sprouts position on which players take turns according to the rules of Sprouts; see Figure 4.3. A player on the move makes a move by connecting two alive spots using a computer mouse. The text block ① indicates

which player is on the move. The alive spots are denoted by black and the dead spots by gray, such as the spots ② and ③, respectively. The drawing of a move obeys the rules of Sprouts, and thus no dead vertex can be connected nor any crossing can be made. After a move is drawn, a new spot is added along that move and the game automatically redraws the position. Further redrawing can be launched with the redrawing button ④. The game ends when there is no remaining move or when a player on the move gives up the game using the button ⑤.

The text block ⑥ displays a useful information that helps to follow the basic strategy from the `Help` dialog. It contains the sum of lives of all vertices in the current position, including the survivors.

4.1.4 Quick play mode

The quick play allows to start a Sprouts game against a human or a computer player either on plain maps that consist only from isolated spots or on custom maps that were drawn in the custom maps tab.

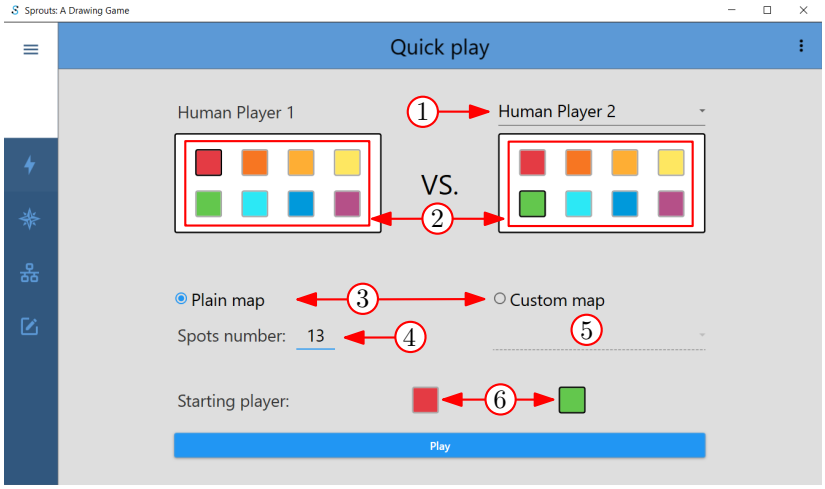


Figure 4.4: The menu of the quick play mode against a human player.

To start a quick game, there is a menu for configuring the settings of the game; see Figure 4.4 for a quick game menu against another human player and Figure 4.5 against a computer opponent. The choice of the opponent type is set using the combo box ①. The colors of human players are set in the tables ② with eight different colors. If the opponent is a computer player, its difficulty is set in a similar table ⑦ that contains four different AI levels (Easy, Medium, Hard, and Insane) with pre-assigned colors.

The radio buttons ③ choose between a plain or a custom map on which the game starts. The radio button of a plain map activates the text box ④ in which the user fills the number of spots of the plain map ranging from 1 up to 20. After the radio button for a custom map is chosen, the combo box ⑤ is activated and the user selects a custom game according to its name from the custom maps tab. The last option is to set the starting player by selection one of the rectangles ⑥ whose colors correspond to the players.

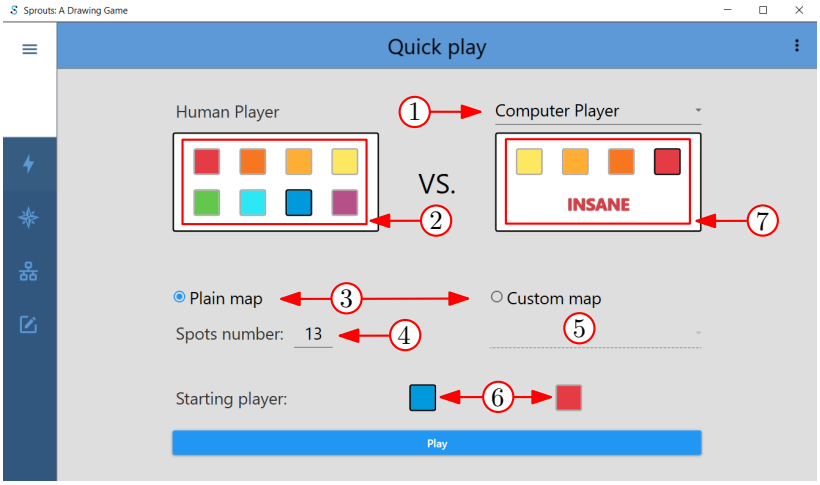


Figure 4.5: The menu of the quick play mode against a computer player.

After all the options are correctly filled, clicking on the play button starts the game with the chosen settings. The game then continues as described in Subsection 4.1.3. When the game ends, a simple dialog in which the user can repeat the game or return back to the quick play menu shows up.

4.1.5 Campaign mode

The campaign is a competitive mode in which players progress through 150 predefined levels against four different AI levels where the most advanced one plays perfectly on each level.

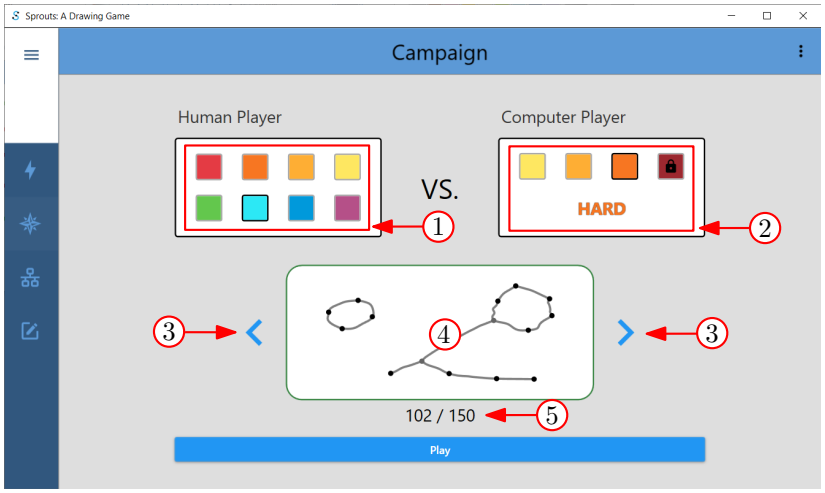


Figure 4.6: The menu of the campaign mode.

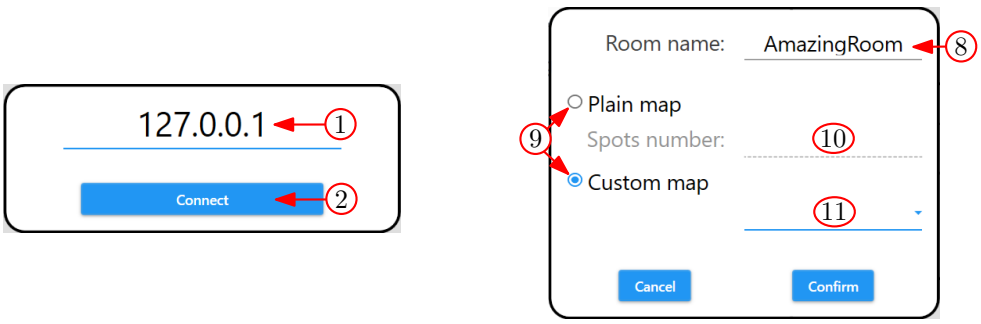
The campaign contains menu that is similar to the menu from the quick play mode; see Figure 4.6. The player can choose his color using the table (1) with eight different colors. The difficulty of a computer opponent (Easy, Medium, Hard, and Perfect AI) is set in the table (2). The AI levels are unlocked for each level according to the previous progress in the campaign. At the beginning, only the first level with all the computer opponents is unlocked. Then, a computer

opponent is unlocked if the AI of at least the same difficulty was beaten in the previous level. After a computer is beaten on the last level, the user receives an achievement according to its difficulty. The level to be played can be set using the arrows ③. The box ④ then contains a view of the chosen level and its number is shown in the text block ⑤.

If an unlocked difficulty is chosen, the play button starts the game. There is no option for selecting the starting player since the player always starts in a winning position and the computer starts in a losing position so that a player can always win even against the perfectly playing computer opponent. The game then continues as described in Subsection 4.1.3. After the game is over, a dialog with two buttons appears. The first button either repeats the same game or it starts the next level against the same AI if it was newly unlocked there. The second button serves for returning back to the campaign menu.

4.1.6 Remote game mode

Our application offers a play with other players across a computer network using the remote game mode. The application connects to the Sprouts server from Section 4.2 using the connection dialog from Figure 4.7. To connect to a server, the user fills the text box ① with server's IP address and then presses the connection button ②. When a connection is being processed, it can be canceled using the same button ②.



The dialog that connects to a server. The dialog creating a game room.

Figure 4.7: The dialogs occurring in the remote game mode.

After a connection is successfully established, the user is moved to the menu containing a list of game rooms that were created by other players on the server and who are waiting for an opponent; see Figure 4.8. The list of game rooms is located in the box ④. Each room consists of a name and a position on which the players will play. A game room can be selected by clicking on it, such as the room ⑤, to view its position in the box ⑥.

The button ⑦ creates a game room and shows the dialog from Figure 4.7 where the user specifies properties of the game room such as the name in the text box ⑧. The room can contain either a plain map or a custom map. This is chosen by the the radio buttons ⑨. Then, the user specifies the number of spots in the text box ⑩ by selecting a number between 1 and 20 in the first case or chooses

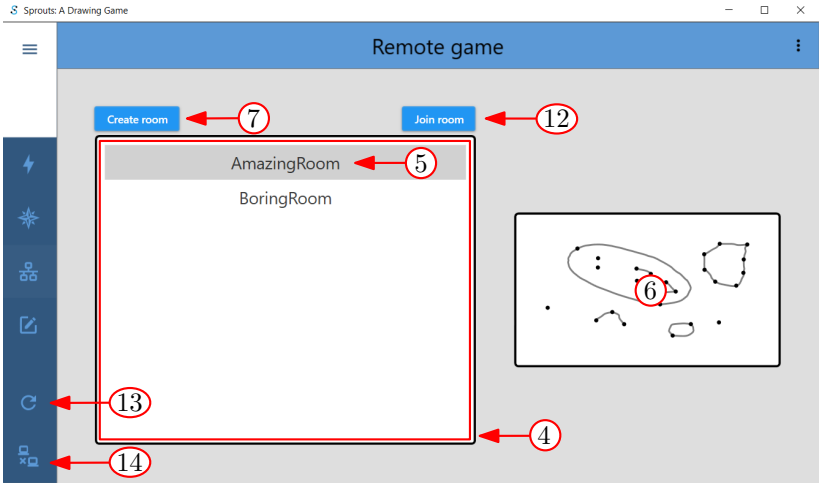


Figure 4.8: The menu with the list of game rooms.

the custom map in the combo box (11) in the same way as in the quick play mode. After a successful creation of a game room, a new waiting dialog shows up, in which the user can cancel waiting by the corresponding button.

Now, we return to the menu with game rooms from Figure 4.8. If the user wants to join a room, he can choose it by clicking on the button (12). It can happen that the joining fails, for example when the room no longer exists. Then, the list of rooms is refreshed only. The user can also refresh the list manually using the button (13). The last button (14) serves for disconnecting from the server.

After the players are interconnected, a remote game starts. We use a special opening called *Swap2* [37] which is a standard opening in *Gomoku* to set fair conditions for both players. In the Swap2 opening, one player chooses the position (the hosting player in our case) and the second player chooses the starting player. However, to make the opening more complex and to try to reduce prepared tactics of the first player, the Swap2 opening also offers an option for the second player to postpone the decision to choose the starting player by making an arbitrary allowed move in the position. The first player must then choose the starting player without any further option of postponing the decision.

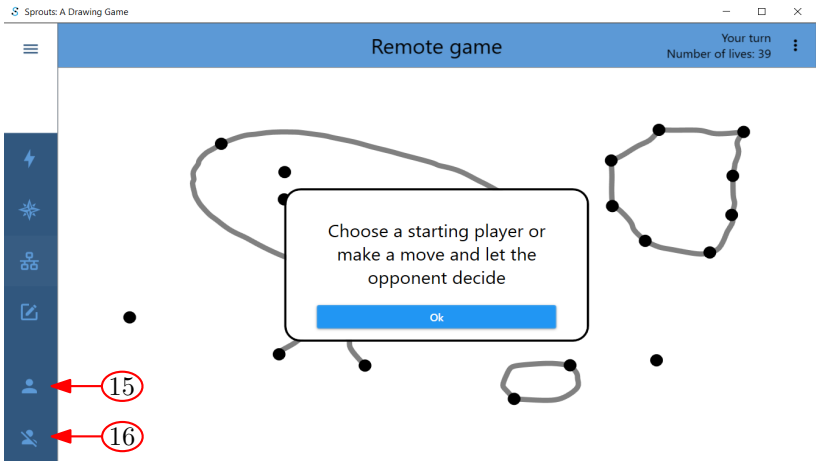


Figure 4.9: The Swap2 opening of a remote game.

To open a remote game with the Swap2 opening, there are two new buttons for choosing the starting player; see Figure 4.9. The button (15) chooses the user as the starting player whereas the button (16) chooses the opponent. Both the buttons become activated after clicking on the confirmation button of the information dialog. If the user wants to postpone the decision, he just draws a move into the position and the move will be passed to the opponent.

After the game is opened, it continues as described in Subsection 4.1.3. The only difference is that only the player on the move can request extra redrawing of a position and that redrawing will be also send to the opponent to keep the positions the same. The game ends when there is no remaining move or when one of the players gives up. Then, there is a dialog that allows to play the same position again if both the players agree or that allows to return to the menu with game rooms.

In exceptional cases, it can happen that the opponent or the server disconnects. This results in the corresponding dialog that returns the user to the menu with game rooms or to the connection menu, respectively.

4.1.7 Custom maps

The quick play and the remote game modes offer to play not only on plain positions but also on custom ones. The tab with custom maps allows to create and manage them.

The tab consists of menu that contains a list (1) of all maps stored by the application; see Figure 4.10. Any map can be chosen by clicking on it, such as the map (2). The box (3) then displays the position of a chosen map. The button (4) removes the chosen map. A new map can be created by clicking on the button (5) that shows the dialog from Figure 4.11. Then, the user fills the text box (6) with a unique name and the text box (7) with a number of initial spots ranging from 1 up to 20 which sets the parameters of the map.

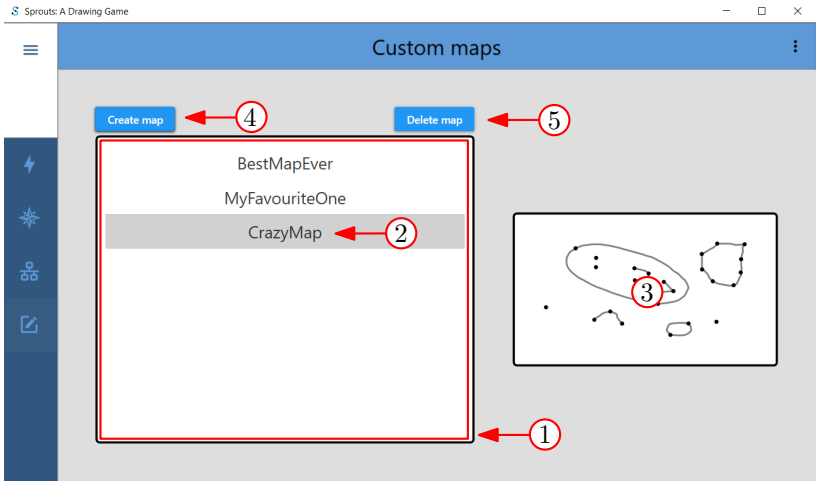


Figure 4.10: The menu of the tab with custom maps.

After filling the dialog, the initial position with the given number of spots is displayed. Now, the user can draw an arbitrary position for custom maps as described in Subsection 4.1.3. The difference is that the edges are drawn with a

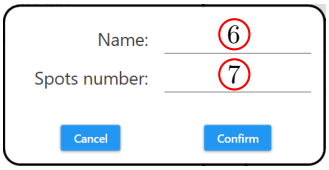


Figure 4.11: The dialog for setting the parameters of a new custom map.

neutral gray color and there is no button for giving up. Instead, there is a button for saving the drawn position.

4.2 Server application

The *Server* application is a console program that is designed to communicate with Sprouts: A Drawing Game applications, and thus it forms the environment for the remote game mode. Its executable file `Server.exe` is located in the `installation\Server` directory in Attachment A. Since it is running on .NET Core 3.1 framework, it is portable between devices that contain the framework; we tested the application on the Windows and Linux operating systems.

Using the application is very simple. It suffices to launch the application on a device that should be the mediator of the network communication and the server then starts listening to any accessible network interface on the port that is defined in `NetworkClasses`; it is the port `20100` in the current version. Then, any client can connect to the server and start the communication according to the communication protocol from Subsection 3.4.2. Each of the received packets is logged to the console output as shown in Figure 4.12. The server can be shutdown by pressing the `Ctrl`+`C` shortcut at any time.

Our Sprouts server is currently running on a Linux server and it is accessible on the IP address `195.113.17.155`. To use his own Sprouts server, the user should make sure that it is accessible in the given network. For example, the used IP address should be visible for the endpoints and there should be no firewall filtering the packets.

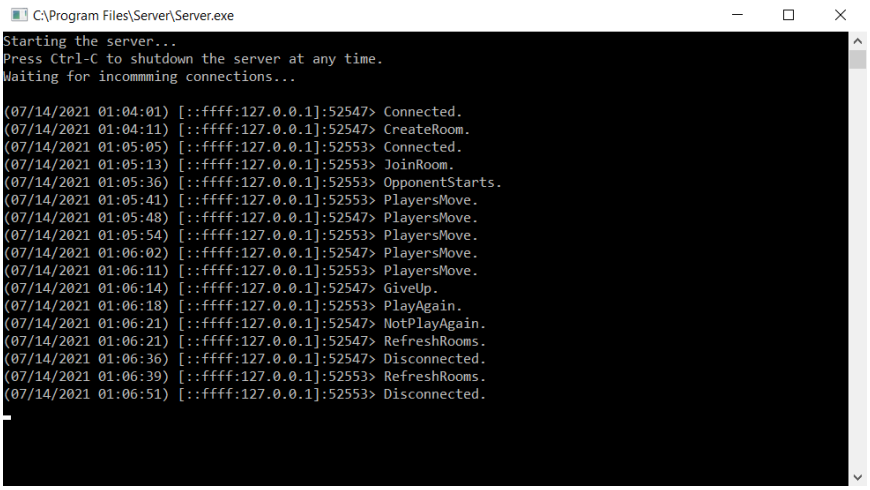


Figure 4.12: The server application.

5. Conclusion

At the end of this thesis, we would like to evaluate all the stated goals from Subsection 1.4 and to also discuss future work that would follow up on the new methods from Chapter 2 or that would extend our application.

5.1 Achieved goals

In Chapter 1, we pointed out that there is almost no solid implementation of Sprouts. Therefore, in Subsection 1.4, we stated six different requirements a solid implementation should have and we decided to create a application that would satisfy all of them. We managed to meet all of the stated goals and we now discuss how exactly they were achieved.

G1 *Free-form input curves*: We decided to support a simple and comfortable way how to draw curves that obey the rules of Sprouts. In our application, the drawings are made simply and intuitively by a computer mouse and there is also a mechanism that automatically checks forbidden crossings which are resolved comfortably during drawing.

G2 *Maintaining positions clear*: Since Sprouts positions tend to degenerate, a mechanism maintaining positions clear needs to be included. Therefore, we introduced a redrawing algorithm that performs very well and is able to finish computations in real-time even on positions with a large number of edges thanks to the use of quadtrees.

G3 *Strong artificial intelligence*: The only implementation of Sprouts that supports a game against a computer opponent is 3Graph by Stefan Reiss [25]. In 3Graph, there is an AI player that plays perfectly on each level from the campaign mode that consists of positions that can be obtained from n-spot positions with n up to 8. Therefore, our goal was to design various levels of a computer opponent where the most advanced one should be also able to play perfectly on pre-trained positions that can be obtained from n-spot positions with n up to 8 and possibly some with $n > 8$. Furthermore, we decided to support games against human or not necessarily perfect computer players on arbitrary positions obtained from n-spot positions for n up to 15 whereas 3Graph only supports positions with n up to 8.

To reach these goals, we first proposed a new spindle method that can draw computer moves on arbitrary positions which, moreover, look naturally and resemble moves drawn by a human player. Since the computation time of the drawing algorithm as well as the redrawing algorithm is small enough, we can afford to support games against human and not necessarily perfect computer players on arbitrary positions obtained from n-spot positions with n up to 20 which is even more than we stated in Goal **G3** and more than twice as much when compared to 3Graph.

To develop various AI levels and mainly the perfectly playing opponent, we applied techniques used by Lemoine and Viennot [6] in their state-of-the-art Sprouts solver and we combined them with our method that trains

a computer opponent to play perfectly on a given position. Using this combination, we created an AI player who plays perfectly on each pre-trained position in the campaign mode that consists of positions obtainable from n-spot positions even up to $n = 11$, exceeding our expectations again.

G4 *User-friendly GUI*: We decided to develop such a user interface of the application that would be highly intuitive even for players who have never played Sprouts before. Therefore, our application consists of simple and intuitive menus and with an attached clear user guide describing the application.

G5 *Remote game*: Our application also meets the goal to support remote games. Besides the possibility of creating game rooms, we also implemented a special game opening to make remote games even more interesting.

G6 *Custom games*: Finally, we also achieve the last goal which is a support of own custom positions that can be played locally against another human or a computer player or remotely against other human players.

5.2 Future work

The newly proposed algorithms from Chapter 2 are still opened for optimizations. For example, the enfoldings of singletons in the spindle method could be chosen more properly to make the computer moves look even more natural. Another option is to employ the distributed computations of outcomes in an implementation of a new Sprouts solver in order to determine the outcomes of new n-spot positions and to tackle the Sprouts conjecture.

Furthermore, our application could serve very well for future work that would explore new properties of Sprouts along with new game strategies. Finally, we would like to modify our program and develop an application for mobile devices since the game of Sprouts is ideal for touchscreens.

Bibliography

[1] J. Nešetřil and J. Matoušek. *Kapitoly z diskrétní matematiky*. Matfyzpress, Praha, 1996.

[2] S. Roberts. *Genius At Play: The Curious Mind of John Horton Conway*. Bloomsbury Press, New York, 2015.

[3] E.R. Berlekamp, J.H. Conway, and R.K. Guy. *Winning ways for your mathematical plays. Vol. 3*. A K Peters, Ltd., Natick, MA, second edition, 2003.

[4] The Complete (?) List of References for the Game of Sprouts. http://gameofsprouts.com/refs.html. Accessed: 2021-05-03.

[5] C. Browne. Algorithms for interactive Sprouts. *Theoret. Comput. Sci.*, 644:29–42, 2016.

[6] J. Lemoine and S. Viennot. Computer analysis of Sprouts with nimbers. In *Games of no chance 4*, volume 63 of *Math. Sci. Res. Inst. Publ.*, pages 161–181. Cambridge Univ. Press, New York, 2015.

[7] M. Gardner. Mathematical games: of sprouts and Brussels sprouts; games with a topological flavour. *Sci. Amer.*, 217:112–115, 1967.

[8] P. Anthony. *Macroscope*. Avon, New York, 1969.

[9] D. Pritchard. *Brain Games: The World's Best Games for Two*. Penguin Books, New York, 1982.

[10] M. Gardner. *Mathematical Carnival*. Penguin Books, New York, 1990.

[11] R. Focardi and F.L. Luccio. A modular approach to Sprouts. *Discrete Appl. Math.*, 144(3):303–319, 2004.

[12] D. Applegate, G. Jacobson, and D. Sleator. Computer analysis of Sprouts. Carnegie Mellon University Computer Science technical report CMU-CS-91-144, 1991.

[13] Sprouts Output Records. http://sprouts.tuxfamily.org/wiki/doku.php?id=records. Accessed: 2021-05-03.

[14] World Game Of Sprouts Association. http://www.wgosa.org/SproutsterWorld.htm. Accessed: 2021-05-03.

[15] The Dot Game That Breaks Your Brain. https://youtu.be/XKsTGdN-4_M. Accessed: 2021-05-03.

[16] J.H. Conway. *On Numbers and Games*. Academic Press, London, 1976.

[17] J. Lemoine and S. Viennot. Analysis of misere Sprouts game with reduced canonical trees. https://arxiv.org/abs/0908.4407, 2009. Accessed: 2021-01-07.

[18] G. Cairns and K. Chartarrayawadee. Brussels sprouts and cloves. *Math. Mag.*, 80:46–58, 02 2007.

[19] P. Alfeld. The Game of Sprouts. http://www.math.utah.edu/ alfeld/Sprouts/index.html, 1999. Accessed: 2021-15-03.

[20] C. Rocchini. The (Computer Version of) Game of Sprouts. http://game-of-sprouts.sourceforge.net/, 2006. Accessed: 2021-22-05.

[21] L.D. Gehrig. Sprouts - A Game of Maths! https://appadvice.com/game/app/spouts-a-game-of-maths/426618463?fbclid=IwAR19mFHqisSybUHpvEpK4s2ZJ2QO74sR1urst7Up6UtebT_dnerpyES8mT0, 2011. Accessed: 2021-15-03.

[22] L.D. Gehrig. Ldgcomputing. http://www.ldgcomputing.com/index.html. Accessed: 2021-15-03.

[23] C.E. Altmann. Sproutsplus. https://apps.apple.com/au/app/sproutsplus/id1490125456#?platform=iphone, 2019. Accessed: 2021-15-03.

[24] G. Egri. Sproutsgame. https://apps.apple.com/uy/app/sprouts-game/id1479115453, 2019. Accessed: 2021-15-03.

[25] S. Reiss. 3Graph. http://www.reisz.de/3graph.htm, 2009. The website is no longer available. Alternative link: https://www.heise.de/download/product/3graph-64693 Accessed: 2021-01-07.

[26] P. Simonetto, D. Archambault, D. Auber, and R. Bourqui. ImPrEd: An Improved Force-Directed Algorithm that Prevents Nodes from Crossing Edges. *Computer Graphics Forum*, 30(3):1071–1080, 2011.

[27] F. Bertault. A force-directed algorithm that preserves edge crossing properties. In *Graph drawing (Štiřín Castle, 1999)*, volume 1731 of *Lecture Notes in Comput. Sci.*, pages 351–358. Springer, Berlin, 1999.

[28] W. Brown and L. Baird. A graph drawing algorithm for the game of Sprouts. In *Proceedings of the 2008 International Conference on Computer Graphics & Virtual Reality, CGVR 2008, July 14-17, 2008, Las Vegas, Nevada, USA*, pages 117–121. CSREA Press, 2008.

[29] Programmer Sought. https://www.programmersought.com/article/95424259627/. Accessed: 2021-04-25.

[30] R.P. Sprague. Über mathematische Kampfspiele. *Tohoku Math. J.*, 41:438–444, 1935–36.

[31] P.M. Grundy. Mathematics and games. *Eureka*, 2:6–8, 1939.

[32] J. Shewchuk. A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator. https://www.cs.cmu.edu/ quake/triangle.html. Accessed: 2021-12-05.

[33] C. Woltering. Triangle.NET. https://archive.codeplex.com/?p=triangle. Accessed: 2021-12-05.

[34] J. Willock. Material Design Themes (4.0.0). https://www.nuget.org/packages/MaterialDesignThemes/4.0.0. Accessed: 2021-07-05.

[35] J. Newton-King. Newtonsoft.Json (13.0.1). https://www.nuget.org/packages/Newtonsoft.Json/13.0.1. Accessed: 2021-07-05.

[36] Microsoft Visual Studio Installer Projects. https://marketplace.visualstudio.com/items?itemName=VisualStudioClient.MicrosoftVisualStudio2017InstallerProjects. Accessed: 2021-07-13.

[37] Gomoku Swap2 Opening. https://en.wikipedia.org/wiki/Gomoku#Swap2. Accessed: 2021-07-10.

A. Attachments

The structure of the electronic attachments.

- **installation** – a directory containing compiled binary files
  - **Server** – a compiled version of the server application
  - **Sprouts** – a win_x64 installer of the Sprouts application
- **src** – a directory with source files of the application
  - **GameModel** – the core project of the implementation that contains many important algorithms and data structures
  - **NetworkClasses** – the auxiliary project for the network communication
  - **Server** – the console application that manages the communication during a remote game
  - **Setup** – the project for producing installers
  - **Triangle.NET** – the external library that is used for triangulations
  - **UserInterface** – the project for implementing the user interface of the application
  - **Sprouts.sln** – a solution file for Visual Studio 2019
- **README.txt** – a file describing the attachments
- **thesis.pdf** – an electronic version of this thesis