A Visual Solver For Fair Division - Adding State-Of-The-Art-Algorithms

Dylan McGrath

Master of Science
Artificial Intelligence
School of Informatics
University of Edinburgh
2024

Abstract

The Fair Slice platform is an educational tool that teaches about envy-free cake-cutting with interactive courses and a resource-splitting tool that uses two simple algorithms, Cut-and-Choose and Selfridge Conway, to return envy-free divisions for two and three agents, respectively. This project aims to enhance the Fair Slice platform through the integration of two efficient state-of-the-art algorithms, Brânzei-Nisan and Hollender-Rubinstein, for three and four agents, respectively, as well as the Piecewise-Constant algorithm, which can find envy-free divisions for any number of agents, albeit less efficiently and with more constraints on valuation function form. The integration of these algorithms required the development and implementation of a new backend to the platform to support their complexity and functionality. Additionally, the project explores the possibility of generalising the Hollender-Rubinstein algorithm for five agents. This work broadens the scope of Fair Slice and contributes to the ongoing development of fair division algorithms in educational contexts.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Dylan McGrath*)

Acknowledgements

I want to acknowledge Andy Ernst whose development of the Fair Slice platform made this project possible.

I also want to acknowledge my supervisor, Dr. Filos-Ratsikas, whose help was invaluable to me over the course of my work.

Table of Contents

Chapter 1

Introduction

This project aims to improve the platform Fair Slice [7], an interactive resource-splitting tool for fair division developed by Andy Ernst, by implementing additional algorithms for envy-free cake-cutting. The platform is designed to teach users about the study of fair cake-cutting via interactive courses and functionality that allows users to input different agents' preferences for the cake via piecewise-linear valuation functions and then returns an envy-free division for the cake by running one of a number of cake-cutting algorithms. Before my contribution, the platform utilised an algorithm for envy-free division between two agents, the Cut-and-Choose method, and between three agents, the Selfridge-Conway method [4]. These methods are simple and are, therefore, implemented in the platform's frontend in TypeScript. The initial state of the website was that it was aesthetically appealing, easy to use, and educational. The algorithms offered left a lot to be desired, however. The motivation behind this project is to bring the quality and choice of algorithms to a level that is in keeping with the quality of the rest of the platform. This will educate users on more complex contemporary algorithms and give them a better understanding of how the area of study looks today.

My planned contribution was implementing a new four-agent algorithm onto the platform, the Hollender-Rubinstein algorithm [10]. As the project developed, this plan was expanded also to include the implementation of the Brânzei-Nisan [6], which is a similar algorithm for three agents, and the Piecewise-Constant algorithm, which is a new algorithm that does not currently have supporting research behind it that works for any number of agents. However, it only works with piecewise-constant valuation functions. The structure of this dissertation is first a description of the implemented algorithms and how they were implemented and then a discussion of their respective runtimes. I also included an additional chapter introducing an idea for a possible five-agent algorithm

that I worked on during the primary work of the project.

Chapter 2

Background

Fair cake-cutting is an area of study for finding divisions of a $[0, 1]$ interval, conceptualised as a cake, for multiple agents with unique valuation functions over that interval such that each agent's assigned slices are deemed fair. There are many kinds of fairness, but the most relevant for our purposes are as follows:

- **Proportionality:** This metric for fairness for n agents necessitates that each agent i is assigned a section of the cake $A_i \subset [0, 1]$ such that their value of this section $v_i(A_i) \geq 1/n$.
- **Envy-Freeness:** This metric for fairness for n agents necessitates that each agent i is assigned a section of the cake $A_i \subset [0, 1]$ such that their value of this section $v_i(A_i) \geq \max_{j \neq i} v_i(A_j)$ for $j \in \{1, \ldots, n\}$.

The metric we will mainly be focusing on for this project is envy-freeness, but it is important to note that all envy-free divisions are proportional by definition.

While this problem may appear abstract, the study of fair division has numerous applications in the real world. Some areas where fair division is imperative are the likes of auctions and elections, two problems where fair division solutions are discussed in some detail by Brams and Taylor [4], two prominent scholars in the field of fair division. More recently, fair division has proved promising in computer science for problems such as resource allocation in shared computer clusters [11] such as the Eddie cluster at the University of Edinburgh.

Steinhaus first introduced fair cake-cutting as an area of study in the field of algorithms [14] in 1948. Specifically, he introduced the concept of proportionality, as defined above. Envy-freeness, that is, methods for division that guarantee all agents receive a slice that is at least tied for their favourite slice, was later popularised by

Gamow and Stern [9] in 1958. The Cut-and-Choose and Selfridge-Conway algorithms are both proportional and envy-free. The idea behind Cut-and-Choose is that Agent 1 cuts the cake into two equal-valued pieces, and Agent 2 chooses their favourite [14]. Agent 1 then receives the remaining slice. This method for envy-free division between two agents is exceedingly simple, but the problem becomes much more complex as more agents are added. The Selfridge-Conway method builds on the Cut-and-Choose method by having agent 1 divide the cake into three equal-valued slices, and the slices are then trimmed and divided by the other players according to a protocol such that an envy-free allocation is achieved in a bounded number of cuts, the maximum number of cuts at most five. These two methods are computationally simple, but Selfridge-Conway, in particular, does not have a pleasant solution with minimum cuts. This problem is only exacerbated when attempts were made to generalise the divide and trim approach to higher number of agents, with Brams and Taylor developing methods that either necessitate theoretically unbounded numbers of cuts or that only divide part of the cake between the agents with some left over [3]. To discuss a solution to this problem, I will first describe the idea of moving knife procedures.

Stromquist first introduced moving-knife procedures in 1980 [15]. They are an approach to envy-free division wherein one or more agents or some third party move a knife or knives across a cake according to some rule to reach a point where the agents agree that the knife positions would result in an envy-free division if they were to cut the cake at these positions. To illustrate such a procedure, consider Austin's representation of the Cut-and-Choose method as a moving-knife procedure [1]. This procedure involves a referee uniformly moving a knife across a cake from the leftmost edge to the rightmost edge. When a point is reached where an agent believes the amount of cake to the left of the knife is equal to the value to the right, that agent, say Agent 1, will say "stop", at which point the cake is cut, and Agent 1 receives the left slice. Agent 2 then gets the right slice. Agent 2 must believe the right slice has at least equal value to the left slice, or they would have said "stop" earlier. Therefore, this must be an envy-free division.

Barbanell and Brams later applied a moving knife procedure to the envy-free division problem for three agents, resulting in a division with at most two cuts, the minimum possible [2]. The procedure, as they described it, was to have a referee uniformly move a knife from right to left across the cake until one agent, say Agent 1, believes the amount of cake to the right of the knife is worth one-third of the total value of the cake. At this point, Agent 1 says "stop" and places markers below the knife and another

marker to the left of this marker such that the two markers divide the cake into three equal-valued slices according to Agent 1's valuations. There are then three possible cases:

1. The division is envy-free
2. Both Agents 2 & 3 prefer slice the left slice
3. Both Agents 2 & 3 prefer slice the middle slice

If the division is envy-free, the cake is cut, and the slices are assigned. Otherwise, the referee holds a knife above the marker immediately to the right of Agent 2 & 3's preferred slice and moves it uniformly to the left. At the same time, Agent 1 holds a knife above the other marker and moves it at a speed that ensures they are indifferent between the two slices not preferred by Agents 2 & 3. At the point where one of either Agent 2 or 3, say Agent 2, is indifferent between their initially preferred slice and one of the remaining two slices, they say "stop" and the cake is cut. Agent 2 is assigned the slice they weakly prefer of the two slices they did not initially prefer, Agent 1 is assigned the other of these two slices, and Agent 3 is assigned the final slice, which they still strictly prefer. While this method is conceptually straightforward, it is a continuous method and, therefore, cannot be implemented as a computer algorithm, as computers can only work with discrete values. Brânzei and Nisan solved this issue [6] in 2017 by utilising the Robertson-Webb (RW) query model[16] and their novel method for discretisation.

The Robertson-Webb query model defines two valuable tools for researching cake-cutting algorithms: eval and cut queries.

- $Cut_i(\alpha)$: Returns the position of the cut y, such that the value of the slice $[0, y]$ for agent i, $v_i([0, y]) = \alpha$.
- $Eval_i(y)$: Returns the value $v_i([0, y])$.

This model assumes the unboundedness and additivity of valuation functions and allows the cut queries to return irrational numbers. This means that the model does not readily apply to the broad monotonicity assumption for valuation functions and cannot immediately be utilised by computer algorithms. Still, with the right discretisation techniques, it is very powerful.

The Brânzei-Nisan algorithm utilises sufficient discretisation methods for eval and cut queries to allow the discretisation of the Barbanell-Brams moving-knife procedure,

which I will describe in Chapter 3. In 2023, Hollender-Rubinstein [10] expanded the ideas of Brânzei-Nisan to develop a discretised computer algorithm for four agents—the first efficient four-agent envy-free cake-cutting algorithm. I will describe their methods in Chapter 4.

The way cake-cutting is handled within the Fair Slice platform involves first defining the agents' valuation functions via inputting piecewise-linear functions in the interactive graph in the resource splitting tool. This interactive graph allows users to input any segments with either uniform (constant) or slanted (linear) valuations from 0 to 10. These are then converted into a list of dictionaries with relevant elements, start and end, which define the location of the segments, and startValue and endValue, which define the values of the segments. Using these values, simple arithmetic methods can be used to find the slice values between two cuts as the area under the defined curves, which forms the basis of any cake-cutting algorithm. The way the algorithms that I implemented in the Python backend were integrated with the website's frontend, as it was written in TypeScript, involved communication between different web addresses where the frontend requested data from the backend web address by first sending the valuation functions as a list of dicts to the backend where it was used to determine an envy-free division. This division was then sent back to the frontend as a JSON object along with the assignments and information that the frontend uses to build a results steps screen that shows the steps the chosen algorithm took to reach an envy-free division.

To see the initial website with just Cut-and-Choose and Selfridge-Conway, visit (`https://fairslice.netlify.app/`) or the Andy Ernst's personal github repo (`https://github.com/AndyCErnst/cake`). For the current iteration of the website, visit (`https://fairslicebeta.netlify.app/`) for the frontend or, to utilise the resource splitting tool, launch from my personal github repo (`https://github.com/dylanmach/cake`) by running "npm start" in the root directory and "flask run" in the backend directory. The updated website is not yet live as I do not have access to Andy Ernst's personal GitHub, where it is deployed.

Chapter 3

The Brânzei-Nisan Algorithm

3.1 Input preprocessing

The necessary preprocessing of the inputted valuation functions is threefold:

1. The valuation function must first be modified such that the cake lies on the interval $[0,1]$. The inputted valuation functions are an interval from 0 to the cake size, being the number of piecewise-linear sections defined by the user. This means that each section's start and endpoint must be divided by the cake size inputted.

2. The valuation function must then be adjusted such that the function is 1-Lipschitz continuous [10] on inputted intervals; in essence,

$$| v_i(a,b) - v_i(a',b') | \leq || a - a' | - | b - b' || .$$

   The simplest method for this was to divide the start and end values of each segment by the max valuation that can be inputted by the platform, in this case 10, and then divide ε by this same factor to ensure its correctness after renormalisation.

3. Finally, the valuation functions must be modified to ensure hungriness or strict monotonicity [6] (that is $v_i(x) > 0 \forall x \in [0,1]$) via the following preprocessing:

$$v_i'(x) = (1 - \varepsilon/2) \cdot v_i(x) + \varepsilon/2.$$

   The hungriness preprocessing can be implemented by replacing the start and endpoints of each section of the inputted agent valuation functions with the new values.

3.2 Eval and Cut Queries

The main tools to be implemented before beginning work on this algorithm are the eval and cut queries. These act much the same as the RW equivalents, but they must be implemented in a discretised manner in order to be used in computer algorithms. Brânzei-Nisan's solution to this discretisation, beginning with eval queries, is as follows

- The valuation functions are approximated via piecewise-constant functions on an ε-grid $\{0, \varepsilon, 2\varepsilon, \ldots, 1-\varepsilon, 1\}$ such that
  - if x is divisible by ε:

$$v_i''([0,x]) = \lceil v_i'([0,x]) \rceil_\varepsilon \tag{3.1}$$

    (Essentially, the value up to the nearest integer multiple of ε). By additivity, we can then find $v_i''([a,b])$, given that both a and b are divisible by ε, by subtraction

$$v_i''([a,b]) = v_i''([0,b]) - v_i''([0,a]). \tag{3.2}$$

  - if x is not divisible by ε, find k such that $k\varepsilon < x < (k+1)\varepsilon$. We know by additivity that

$$v_i''([0,x]) = v_i''([0,k\varepsilon]) + v_i''([k\varepsilon,x]). \tag{3.3}$$

    As $k\varepsilon$ is divisible by ε, $v_i''([0,k\varepsilon])$ can be found via (3.1). For $v_i''([k\varepsilon,x])$, interpolation is used such that

$$v_i''([k\varepsilon,x]) = \frac{x-k\varepsilon}{\varepsilon}\, v_i''([k\varepsilon,(k+1)\varepsilon]). \tag{3.4}$$

    Using these functions, $v_i''([a,b])$ is defined for any $[a,b] \subseteq [0,1]$[5].

When implementing the modified valuation function in the platform, there is no necessity to change the preprocessed valuation functions as the modifications can be performed as the algorithm is running.

Cut queries are then implemented as a binary search of eval queries, such that if you pass in an initial cut position a, value α, and whether the initial cut is an end cut or start cut of a slice, it will run a binary search of eval queries until both the upper and lower bounds of the returned cut lie within $\frac{\varepsilon}{2r}$ of one another where r is stated to be the amount of rounds of communication to find an envy-free division [5]. In this case r

was chosen to be 100, though it could have been chosen to be smaller as there cannot be more than 52 rounds of the main binary search loop before the difference between the inspected cut bounds is less than machine precision (2^{-52} is machine precision in python).

These two methods form the foundation of the algorithm, and all future methods heavily utilise them, so it was essential that they were written in the most computationally efficient way possible. The formulae described above accurately represent how the code was implemented in the platform.

3.3 Implementation of the Algorithm

Step 1: We first need to find the unique cut positions for each agent for which one-third of the cake lies to the right of said cut. These cut positions are guaranteed to be unique because of the hungriness modification made to the valuation functions. The agent who returns the rightmost cut is identified (this agent will hereafter be referred to as Agent 1), and we then find the division for this agent that splits the cake into thirds [6]. This is the equipartition. Because of the additive valuation assumption, this can be found by simply taking the value of the entire cake for Agent 1 and taking cut queries for one-third of this value.

Step 2: Once we have the equipartition, we check if this is itself an ε-envy-free division. If so, we terminate the algorithm here and return the relevant cut positions. If not, we move on to the next step. It is clear by construction of the equipartition that both Agent 2 & 3 prefer either slice 1 or slice 2. A check is run for which cases are true, and the algorithm runs the following corresponding step [6].

Step 3a: If slice 1 is preferred by Agent 2 & 3, the next step is to run a binary search over the left cut. The constraints for this binary search are as follows:

- The lower bound for the left cut position, l_{lower}, is the left bound of the cake, 0. This position for the left cut is guaranteed to return a division where Agent 2 & 3 both prefer slice 2 or 3.
- The upper bound for the left cut position, l_{upper}, is the left cut of the equipartition. This position for the left cut is guaranteed to return a division where Agent 2 & 3 both prefer slice 1.
- For each inspected left cut in the binary search, the right cut, r, is identified such that Agent 1 is indifferent between slices 2 & 3.

These constraints give two assurances. Agent 1 always weakly prefers a slice that is not slice 1, and, by the intermediate value theorem, there must be a point in this binary search where at least one of the other agents will be indifferent between slice 1 and another slice. This guarantees that this binary search will reach a division which is ε-envy-free. The implementation of the binary search, aided by a graph utilised in the Fair Slice platform interactive course, is as follows:

1. Find the midpoint of the left cut bounds, l_{mid}, such that Agent 1's valuations $v_1''([l_{lower}, l_{mid}]) = v_1''([l_{mid}, l_{upper}])$.
2. Find the right cut, r, such that Agent 1's valuations $v_1''([l_{mid}, r]) = v_1''([r, 1])$.
3. If an ε-envy-free division exists at the division $\{l_{mid}, r\}$, terminate the algorithm and return the corresponding cuts. If not, if both Agent 2 & 3 prefer slice 1, set $l_{upper} = l_{mid}$ and continue. Otherwise set $l_{lower} = l_{mid}$ and continue [6].

An example of these algorithm steps as they are returned on Fair Slice is given below.

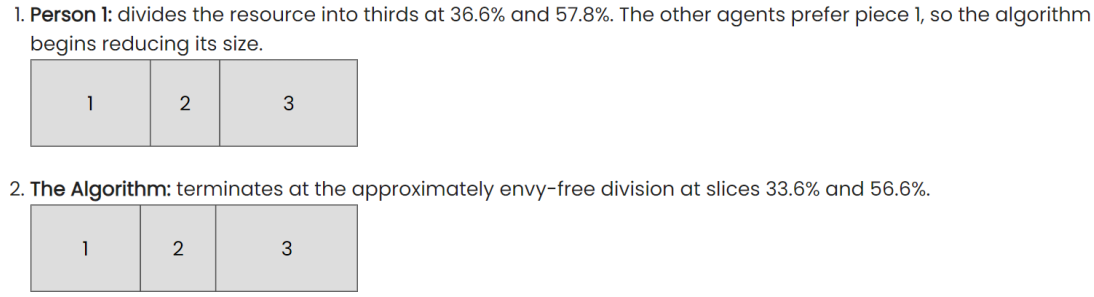

Figure 3.1: Brânzei-Nisan steps for slice 1 preferred case

The main computational problem presented in this implementation is that of a cut that returns two slices of equal value. For this, my solution was to leverage the additive valuation assumption similarly to how it was leveraged in finding the equipartition. Looking at step 1 of this implementation specifically, I took the value $x = v_1''([l_{lower}, l_{upper}])$ and then utilised a cut query, $cut(l_{lower}, x/2)$, to find l_{mid}.

Step 3b: If slice 2 is preferred by Agent 2 & 3, the next step is running a binary search over the right cut. The constraints for this binary search are as follows:

- The lower bound for the right cut position, r_{lower}, is the midpoint of the cake according to Agent 1 (i.e. the cut that satisfies $v_i''([0, r_{lower}]) = v_i''([r_{lower}, 1])$. This position for the right cut is guaranteed to return a division where Agent 2 & 3 both prefer slice 1 or 3 (given the corresponding left cut).

- The upper bound for the right cut position, r_{upper}, is the right cut of the equipartition. This position for the right cut is guaranteed to return a division where Agent 2 & 3 both prefer slice 2 (given the corresponding left cut).
- For each inspected right cut in the binary search, the left cut, l, is identified such that Agent 1 is indifferent between slices 1 & 3.

These constraints offer the same assurances as in step 3a with respect to slice 2. The implementation of this binary search, aided by a graph utilised in the Fair Slice platform interactive course, is as follows:

1. Find the midpoint of the right cut bounds, r_{mid}, such that Agent 1's valuations $v_1''([r_{lower}, r_{mid}]) = v_1''([r_{mid}, r_{upper}])$.
2. Find the left cut, l, such that Agent 1's valuations $v_1''([0, l]) = v_1''([r_{mid}, 1])$.
3. If an ε-envy-free division exists at the division $\{l, r_{mid}\}$, terminate the algorithm and return the corresponding cuts. If not, if both Agent 2 & 3 prefer slice 2, set $r_{upper} = r_{mid}$ and continue. Otherwise set $r_{lower} = r_{mid}$ and continue [6].

Computing the cut positions in this implementation can be done using similar methods as described in step 3a, though bisection is not necessary in step 2 of this implementation.

An example of these algorithm steps as they are returned on Fair Slice is given below.

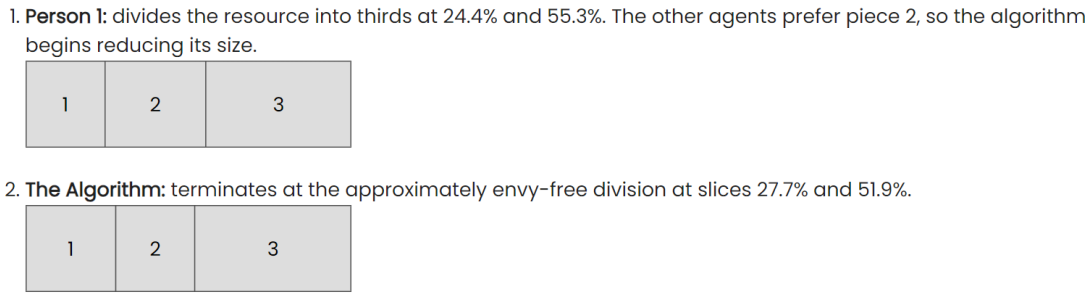

Figure 3.2: Brânzei-Nisan steps for slice 2 preferred case

With the conclusion of these steps, an ε-envy free division will be returned. Because Brânzei-Nisan wrote this algorithm with consideration for the Robertson-Webb query model [16], which does not assume boundedness, there is a further step after both steps 3a and 3b that will run if an envy-free division is not returned once the relevant slice is worth less than ε to Agent 1. At this point, it will start reducing this slice value relative

to Agent 2's until an ε-envy free division is found [6]. This is not necessary for our purposes, as the Fair Slice platform exclusively takes in bounded valuation functions. This ensures that reducing the value of the relevant slice relative to Agent 1's valuations must eventually return an envy-free division as a slice an ε-wide interval cannot be worth more than ε to any agent by the 1-Lipschitz preprocessing of the valuation functions.

It is also worth noting that while the binary search as it is implemented will tend towards indifference between the relevant slice and some other slice for at least one agent (the endpoint of the Barbanel-Brams algorithm), it is unnecessary to run it until it reaches this conclusion if it finds an ε-envy-free division beforehand. Any ε-envy-free division will do. This fact helps improve the runtime of this specific algorithm, but that will not be the case in future algorithms discussed.

The chosen ε value for this algorithm and the Hollender-Rubinstein algorithm is 0.00025 for reasons that will be discussed in the next chapter.

Chapter 4

The Hollender-Rubinstein Algorithm

4.1 Eval and Cut Queries

We begin by performing the same input preprocessing as in the Brânzei-Nisan algorithm with the exception of the hungriness preprocessing, which is done as the algorithm runs as will be described below. Next, we again must implement discretised eval and cut queries. The discretisation in this algorithm is similar to the Brânzei-Nisan algorithm but notably more complex.

- Ensure the eval queries are strictly hungry (according to a different but equivalent definition $v_i([a,b]) < v_i([a',b']) \,\forall\, [a,b] \subsetneq [a',b']$ [10]) for all agents i via the following modification:

  $$v_i'([a,b]) = v_i([a,b])/2 + \varepsilon \mid b-a \mid \in [0,1]$$

  [10] This satisfies strict hungriness as a result of the monotonicity assumption (in fact, the Fair-Slice valuation functions are additive, a special case of monotone) as shown below:

  $$\begin{gathered} v_i([a,b]) < v_i([a',b']) \\ v_i([a,b])/2 + \varepsilon \mid b-a \mid < v_i([a',b'])/2 + \varepsilon \mid b'-a' \mid \\ (v_i([a,b]) - v_i([a',b']))/2 < \varepsilon(\mid b'-a' \mid - \mid b-a \mid) \end{gathered}$$

  By monotonicity of valuations ($v_i([a,b]) \leq v_i([a',b']) \,\forall\, [a,b] \subseteq [a',b']$) the LHS must be less than zero and by definition the RHS must be greater than zero, so this modified valuation function is strictly hungry.

- Then, the valuation functions are approximated via piecewise-linear functions on an ε-grid $\{0, \varepsilon, 2\varepsilon, \ldots, 1-\varepsilon, 1\}$. The ε-interval of the start cut a is identified and upper and lower bounds are identified as $\underline{a} = \varepsilon k \leq a \leq \varepsilon(k+1) = \bar{a}$ and the same process is done to the end cut b to find $\underline{b}$ & $\bar{b}$. If $\bar{a} - a \leq b - \underline{b}$ let

$$v_i''([a,b]) = \frac{(\bar{a}-a)-(b-\underline{b})}{\varepsilon} v_i'([\underline{a},\underline{b}]) + \frac{b-\underline{b}}{\varepsilon} v_i'([\underline{a},\bar{b}]) + \frac{a-\underline{a}}{\varepsilon} v_i'([\bar{a},\underline{b}]). \tag{4.1}$$

  Otherwise, let

$$v_i''([a,b]) = \frac{(b-\underline{b})-(\bar{a}-a)}{\varepsilon} v_i'([\bar{a},\bar{b}]) + \frac{\bar{a}-a}{\varepsilon} v_i'([\underline{a},\bar{b}]) + \frac{\bar{b}-b}{\varepsilon} v_i'([\bar{a},\underline{b}]). \tag{4.2}$$

  [10]

These modifications are again performed as the algorithm is running, and the valuation functions retrieved from the user inputs need not be changed as preprocessing. Each eval query utilised in the algorithm uses, at most, three intermediate eval queries.

One interesting difference between this discretisation and the discretisation performed by Brânzei and Nisan is the lack of subtraction between eval queries on intervals whose intersection is the interval of interest. This results from the Hollender-Rubinstein algorithm finding ε-envy-free divisions for monotone valuation functions, not just additive ones. This makes the algorithm more powerful but means that the implementation can no longer make use of some of the valuable shortcuts that aided the implementation of the Brânzei-Nisan algorithm [10]

Cut queries can then be implemented as a binary search of eval queries like in the Brânzei-Nisan algorithm [10]. While the algorithm can utilise caching of the intermediate eval queries once the ε-interval, my initial implementation did not utilise this fact, so I will discuss the specifics of this approach later in the paper.

4.2 Initial Implementation of the Algorithm

The implementation of this algorithm is conceptually similar to that of the Brânzei-Nisan algorithm but with a few key differences. As previously mentioned, the broader assumption of monotone valuation functions means that the divisions satisfying particular prerequisites will require more computation than in the Brânzei-Nisan algorithm. The Hollender-Rubinstein algorithm is also parameterised over Agent 1's value for their

preferred slice, α. This allows the algorithm to test different cases in each loop where Brânzei-Nisan had to identify one of two cases (either Agents 2 & 3 preferred slice 1 or 2) before the algorithm began working within the identified case exclusively. My initial implementation is described below.

Step 1: We first need to find the unique equipartition for Agent 1 [10]. This is done by finding the unique positions of the three cuts via independent investigation. Because this initial implementation does not utilise caching of intermediate eval queries, it is not strictly necessary to find the cut locations independently. Still, I implemented it this way to make the code easier to adapt later. This fact is true for all subsequent computations of multiple cut locations.

- The left cut location is found via binary search over the range $[0, 1]$. For each inspected left cut l, find the middle cut m such that $v_1''([l, m]) = v_1''([0, l])$ and subsequently the right cut r such that $v_1''([m, r]) = v_1''([0, l])$. This is achieved via successive cut queries. Then, if $v_1''([r, 1]) > v_1''([0, l])$, the updated left cut must move to the right. Otherwise, it must be moved to the left. The right cut can be found analogously.

- The middle cut is again found via binary search over the range $[0, 1]$. For each inspected middle cut m, find the left cut l such that $v_1''([0, l]) = v_1''([l, m])$ and, likewise, the right cut r such that $v_1''([m, r]) = v_1''([r, 1])$. This is achieved via what will be described as a bisection cut query, which is found via a binary search of cut locations between the start and end cut passed in (e.g. 0 & m for the location of l). Then if $v_1''([0, l]) > v_1''([r, 1])$, the updated middle cut must move to the left. Otherwise, it must move to the right.

The criterion for how to update the cut positions leverages the monotonicity and hungriness of the valuation functions. i.e. if l moves to the right, $v_i''([0, l])$ must increase and $v_i''([l, m])$ must decrease. Therefore, m must move to the right to ensure $v_i''([0, l]) = v_i''([l, m])$. This reasoning can be extended to r and utilised to inform how to update cut locations in binary searches according to given conditions.

Step 2: Once we have the equipartition, we check if this is itself an ε-envy-free division. If so, we terminate the algorithm here and return the relevant cut positions. If not, we move on to the main body of the algorithm.

Step 3: The main body of the algorithm is a binary search parameterised over Agent 1's value for their preferred slice, α. Hollender-Rubinstein defines an invariant with two conditions, A & B, for which at least one of the conditions must hold at the α

unique to the equipartition and neither condition holds at $\alpha = 1$ by hungriness. The only exit condition of the invariant is an ε-envy-free division. This informs the structure of the body wherein if the invariant holds at the inspected α, the lower bound $\underline{\alpha}$, initially defined as the α unique to the equipartition, is updated such that $\underline{\alpha} = \alpha$. Otherwise, the upper bound $\bar{\alpha}$, initially defined as 1, is updated such that $\bar{\alpha} = \alpha$. By construction, this binary search will converge to an α that gives an ε-envy-free division. What remains is to describe the implementation of the invariant check.

Condition A: This condition is analogous to the cases discussed in the Brânzei-Nisan algorithm. It states that, given Agent 1 values three slices at α, the remaining slice is preferred by two or more of the other agents. It is also imperative that Agent 1 does not prefer this remaining slice [10]. In the case where slice 2 is the slice of interest, the slice locations are found via cut queries. l is found such that $v_1''([0, l]) = \alpha$, r is found such that $v_i''([r, 1]) = \alpha$ and then this value is used to find m such that $v_1''([m, r]) = \alpha$. It is then necessary to check that the returned division is valid (i.e. $l \leq r$ as there is no reason this cannot be the case with this implementation). If it is not, the algorithm moves on to one of the following cases. If the division is valid, the algorithm checks to see if at least two agents prefer the relevant slice within ε. If it is, $\underline{\alpha}$ is updated and the loop continues. If not, the algorithm moves on to another case. The other cases can be checked using similar or equivalent methods.

Below is an example of the Hollender-Rubinstein steps terminating at condition A as they are returned on Fair Slice.

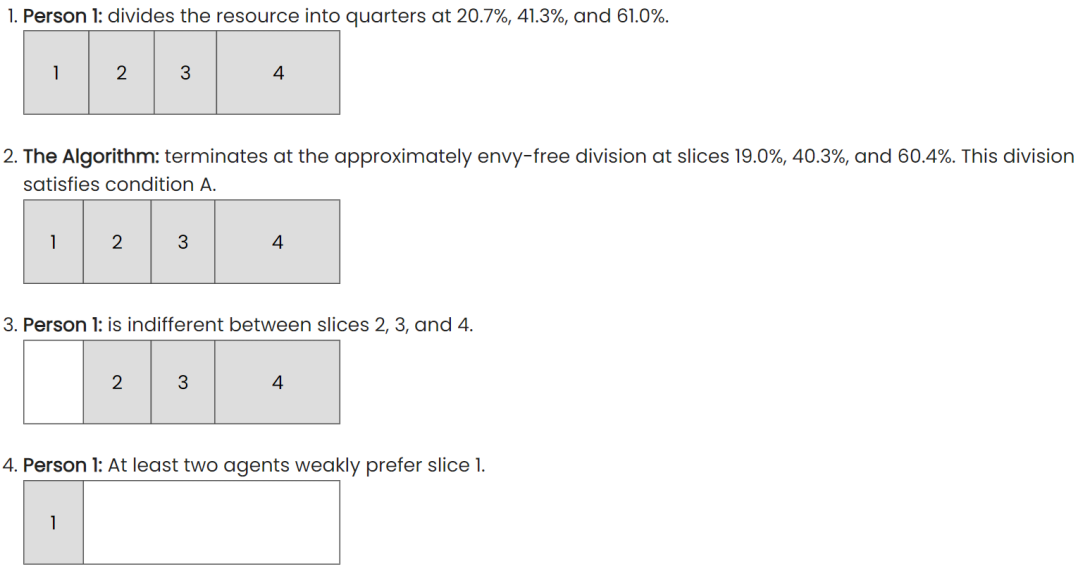

Figure 4.1: Hollender-Rubinstein steps terminating at condition A

Condition B: This condition states that given Agent 1 values two slices at α, the remaining slices are each preferred by two or more of the other agents. It is again imperative that Agent 1 does not prefer either of the remaining slices [10]. Given that there are only three other agents and two slices, it is also necessary that at least one agent is indifferent between the remaining slices for the condition to hold. This fact is leveraged to find the divisions for each case. There are 3 cases for condition B, which I will now describe.

1. **Relevant slices are adjacent:** For the representative case where the relevant slices are 2 and 3, l and r are found via cut queries from the outer edges 0 and 1 of value α according to Agent 1's valuation. We then iterate over each agent we will describe as indifferent and find m via a bisection cut query between l and r according to the indifferent agent's valuation. After checking that the returned division is valid, the algorithm checks that the two slices $[l, m]$ and $[m, r]$ are each preferred by at least two agents. If they are, $\underline{\alpha}$ is updated and the loop continues. If not, the algorithm proceeds to check the following cases. The other cases are handled similarly.

2. **Relevant slices are one apart:** For the representative case where the relevant slices are 1 and 3, r is found via a cut query from the rightmost edge 1 of value α according to Agent 1's valuation. We then iterate over each indifferent agent and find l and m via binary searches. For l, the binary search has bounds $[0, r]$ and for each investigated l, we find m via a cut query with start cut l of value α according to Agent 1's valuation. The bounds for l are then updated analogously to how they were updated when finding the equipartition with the aim that $v_i''([0, l]) = v_i''([m, r])$ where i is the indifferent agent. m is found analogously. After checking that the returned division is valid, the algorithm checks that the two slices $[0, l]$ and $[m, r]$ are each preferred by at least two agents. If they are, $\underline{\alpha}$ is updated and the loop continues. If not, the algorithm proceeds to check the following cases. The other case is handled similarly.

   Relevant slices are two apart: This case is handled similarly to the case where the relevant slices are one apart, so it will not be described in detail as there are no new implementation challenges. Like the previous cases, if the case holds, $\underline{\alpha}$ is updated and the loop continues. If not, given that this is the last case to check, we know the invariant does not hold for this α and $\bar{\alpha}$ is updated before continuing the loop.

Below is an example of the Hollender-Rubinstein steps terminating at condition B as they are returned on Fair Slice.

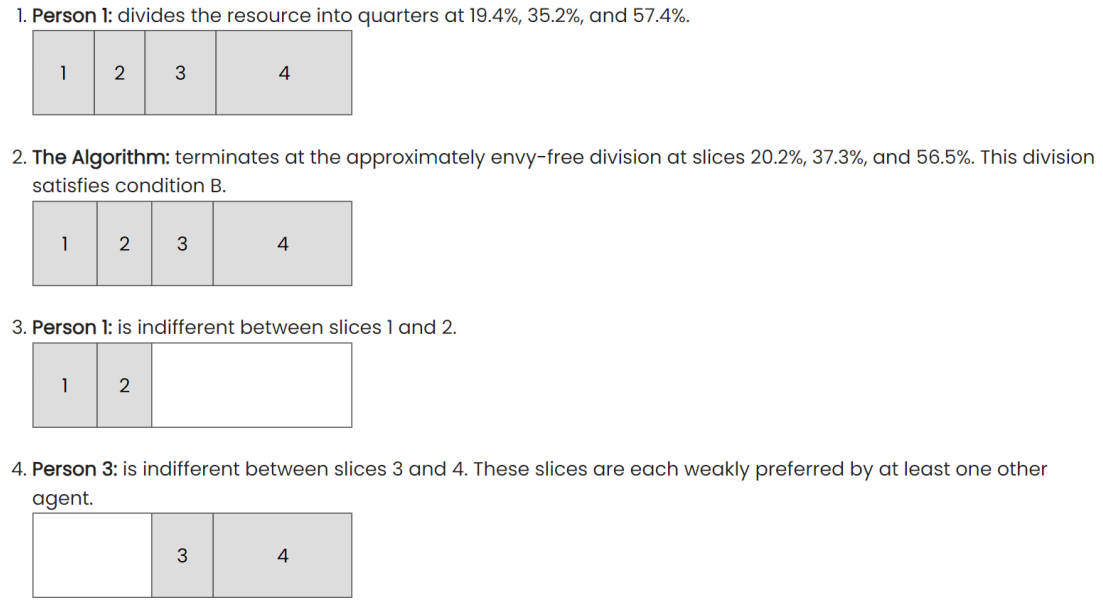

Figure 4.2: Hollender-Rubinstein steps terminating at condition B

One interesting feature of the algorithm is that this binary search over α continues until $|\bar{\alpha} - \underline{\alpha}| < \varepsilon^4/12$ to ensure an ε-envy-free division is returned for the original, unmodified valuation functions. An observation about this tolerance is that it necessitates an ε value that, in some cases, could be prohibitively large. In theory, the algorithm can work for whichever ε one may desire; when taking machine precision into account, the minimum ε is ~ 0.000227. This gets even larger when considering necessary preprocessing to ensure the valuation functions are 1-Lipschitz, ε is divided by whatever constant the valuation function is divided by. This is not a problem for the Fair Slice platform and is likely not an issue for most applications, but it is worth noting as a possible improvement area.

4.3 Slice Assignment

The algorithm, as described by Hollender and Rubinstein [10], is only intended to find an ε-envy-free division. It does not offer any method for assigning the slices once the division is finalised. This presents an interesting problem. Of course, any division that satisfies ε-envy-freeness would be valid, but it is essential to remember that this algorithm is to be implemented in a public resource splitting calculator, so we must

also prioritise user experience as much as possible. Suppose the algorithm assigns slices in an ε-envy-free manner that is not exactly envy-free when an exactly envy-free assignment exists. In that case, this may result in users who are less knowledgeable in the intricacies of the algorithm believing it has been implemented incorrectly. To avoid such scenarios as much as possible, I devised an algorithm for slice assignments that should guarantee exact envy-freeness if such an assignment exists and, if not, will maximise utility for an ε-envy-free division. My first attempt at such an algorithm used a greedy approach, which is described below.

Step 1: Begin with the four slices and four agents. We initialise the algorithm by finding the values of the slices according to each agent's valuations and storing them in a 4×4 array. We then enter the main body of the loop.

Step 2: The algorithm begins a loop that, in each iteration, finds each agent's first and second favourite slices, as well as their difference in valuations. It then identifies the largest valuation difference and the associated agent and assigns that agent their favourite slice. This agent and slice are removed, and the loop continues with three agents and three slices. The loop continues until only one agent remains. At this point, they are assigned the remaining slice.

This algorithm was flawed, however. The table below illustrates this flaw in the three-agent case.

	Slice 1	Slice 2	Slice 3
Agent 1	$0.4+\varepsilon$	0.4	$0.2-\varepsilon$
Agent 2	$1/3+0.55\varepsilon$	$1/3$	$1/3-0.55\varepsilon$
Agent 3	$1/3+0.4\varepsilon$	$1/3+0.2\varepsilon$	$1/3-0.6\varepsilon$

Table 4.1: Slice Values for Three Agents

Given the above agent values for an ε-envy-free division, in the first iteration of the loop, Agent 1 would have the highest difference between their top two slice values (For slices 1 and 2) of ε. This leads to Agent 1 being assigned Slice 1. Agent 1 and slice 1 are removed, and the new slice values look like this.

	Slice 2	Slice 3
Agent 2	$1/3$	$1/3-0.55\varepsilon$
Agent 3	$1/3+0.2\varepsilon$	$1/3-0.6\varepsilon$

Table 4.2: Slice Values for Remaining Two Agents

Now Agent 3 has the largest difference of 0.8ε and is assigned Slice 2. Agent 2 is then assigned Slice 3. But, looking at table (4.1), we can see that the difference between the value of their favourite slice, Slice 1, and their assigned Slice, Slice 3, is 1.1ε. This is more than ε; therefore, this assignment is not ε-envy-free.

To remedy this, I decided to implement an assignment that utilises Hungarian matching via the scipy package in Python [12]. This algorithm takes in a two-dimensional matrix and returns an assignment that minimises the cost. To apply this algorithm to the cake-cutting problem, I first identify all possible slices that can be assigned to each agent (All slices valued within ε of their favourite slice). Then, I assign costs to these slices for each agent equal to the difference between their value and that agent's favourite slice (their favourite slice has cost zero). All slices that cannot be assigned to that agent have cost 10000 to ensure the algorithm cannot assign them. In the case of table (4.1), the cost matrix would look like this:

0	ε	10000
0	0.55ε	10000
0	0.2ε	ε

Table 4.3: Cost Matrix for Three Agents

This would result in Agent 1 being assigned Slice 1, Agent 2 being assigned Slice 2, and Agent 3 being assigned Slice 3, for a total cost of 1.55ε. Note that if random assignment of viable slices was employed, Agent 1 could be assigned Slice 2, and Agent 2 could be assigned Slice 1, but this would not maximise utility. This assignment satisfies the initial goal of maximising utility without violating ε-envy-freeness. If we used Hungarian matching without making the cost of slices that cannot be assigned to agents equal to some large value, it would give the same assignment as the original assignment algorithm, which shows that just minimising cost with respect to value differences is not sufficient.

4.4 Runtime Improvements

What follows are changes to the initial algorithm implementation that Hollender-Rubinstein did not specify to improve runtime. It is essential to state that these changes are not contrary to how Hollender-Rubinstein described the algorithm in the original paper [10] but are instead things that were not explicitly stated in the paper but may

have been implied. The runtime could have been further improved by leveraging the additive nature of the Fair Slice valuation functions, but it was important to prioritise the algorithm's integrity by not deviating from the explicit implementation choices from the paper.

4.4.1 Eval Queries

My first change was how initial eval queries are handled in the code before modification. How eval queries were initially implemented was in line with how they are defined in the Robertson-Webb query model [16]. Here, eval queries are defined as the value of a slice between the left boundary, 0, and a cut position x. Because the Robertson-Webb query model assumes additivity, you can find the between two slices by subtraction of eval queries. This is how I initially constructed the initial eval queries in my code. Because Fair Slice takes inputs as piecewise-linear valuation functions, eval queries involve iterating over each segment of the valuation function and taking the full or partial area under the curve depending on the cut location relative to the endpoint of the segment. The problem with this implementation was that it necessitated calculating the values of the segments to the left of the first cut twice when they were not necessary to calculate. These unnecessary calculations take a very small amount of time, but when the algorithm necessitates $O(log^3(1/\varepsilon))$ eval queries, it amounts to quite a large inefficiency in the code. The new implementation works similarly, but it does not utilise the subtraction of two eval queries from 0 to the start and end cut, respectively. Instead, it only finds the area of the full and partial segments between the start and end cut.

4.4.2 Binary Searches of Eval Queries

The second change was how binary searches of eval queries were handled in the code. In the initial implementation, to find exact cut locations in a division, I utilised an inefficient binary search that continued until the upper and lower bounds of the cut were 1e-15 apart. The reason for this inefficiency was that, in Hollender-Rubinstein's paper, they described these binary searches as finding the ε-interval in the grid where the cut lies, at which point the exact location can be found "using a constant number of queries". The intended method implied by this statement was not initially clear to me, so I wrote the code so that it could easily be optimised later when I needed to implement their intended method but could still move forward with the rest of the algorithm in the meantime. This entailed adding breaks to the binary searches once an ε-interval was

identified and running binary searches for each cut location in a specified division such as the equipartition.

The intended method leverages the fact that once the ε-interval the cut lies on is identified, all future iterations of the binary search will utilise the same intermediate eval queries by definition. This means that these intermediate eval queries can be cached and reused for all future iterations of the binary search. This reduces the necessary computation by a large margin. In fact, for simple cases such as cut queries and bisection cut queries, it is unnecessary to continue the binary search as the cut position can be found via manipulation of the value query formula. We can do this because, looking at equations (4.1) and (4.2), the only variable is the cut we are looking for. The initial cut, expected value, and all bounds are known values. Below, the algebraic manipulation to find the cut b is detailed for the equation (4.1).

$$v_i''([a,b]) = \frac{(\bar{a}-a)-(b-\underline{b})}{\varepsilon} v_i'([\underline{a},\underline{b}]) + \frac{b-\underline{b}}{\varepsilon} v_i'([\underline{a},\bar{b}]) + \frac{a-\underline{a}}{\varepsilon} v_i'([\bar{a},\underline{b}])$$

$$\frac{b-\underline{b}}{\varepsilon} v_i'([\underline{a},\underline{b}]) - \frac{b-\underline{b}}{\varepsilon} v_i'([\underline{a},\bar{b}]) = \frac{\bar{a}-a}{\varepsilon} v_i'([\underline{a},\underline{b}]) + \frac{a-\underline{a}}{\varepsilon} v_i'([\bar{a},\underline{b}]) - v_i''([a,b])$$

$$(b-\underline{b})\frac{v_i'([\underline{a},\underline{b}]) - v_i'([\underline{a},\bar{b}])}{\varepsilon} = \frac{\bar{a}-a}{\varepsilon} v_i'([\underline{a},\underline{b}]) + \frac{a-\underline{a}}{\varepsilon} v_i'([\bar{a},\underline{b}]) - v_i''([a,b])$$

$$b-\underline{b} = \frac{(\bar{a}-a)\, v_i'([\underline{a},\underline{b}]) + (a-\underline{a})\, v_i'([\bar{a},\underline{b}]) - \varepsilon v_i''([a,b])}{v_i'([\underline{a},\underline{b}]) - v_i'([\underline{a},\bar{b}])}$$

$$b = \frac{(\bar{a}-a)\, v_i'([\underline{a},\underline{b}]) + (a-\underline{a})\, v_i'([\bar{a},\underline{b}]) - \varepsilon v_i''([a,b])}{v_i'([\underline{a},\underline{b}]) - v_i'([\underline{a},\bar{b}])} + \underline{b}$$

Once this algebraic manipulation is complete, the algorithm checks that the returned b is between $\underline{b}$ and $\bar{b}$ and also that $\bar{a}-a \leq b-\underline{b}$. The first check is necessary because we know the ε-interval b must lie on for this to be a valid cut location. The second check is necessary because this value for b is only valid if its position satisfies the prerequisite for the first variant of the eval query formula, equation (4.1). This second check presents a more significant issue. Because we do not know the position of the cut before finding it algebraically, we do not know which value query variant we should be manipulating. This means we must check both in the worst case. This is not as much of

a problem in the case of cut queries where there is only one slice value to consider, but consider the case of the equipartition, which has four slice values to consider. To solve it algebraically, one must exhaustively consider 16 (2^4) different algebra combinations in the worst case, all of which involve at least three simultaneous equations to find the three cut positions. This is a lot of code and increases the chances of mistakes in one or more cases that could go unnoticed in testing. For this reason, I decided only to handle cut and bisection cut queries (which have four combinations) algebraically and handle other binary searches via caching. Implementing algebraic solutions for the other binary searches is undoubtedly one method to improve the implementation of the four-agent algorithm as it currently exists within Fair Slice, and it would be interesting to see how much it would improve the runtime as an area for future work.

Continuing the binary search for cases other than simple cut queries and bisection cut queries has its own unique problems, though. Considering the equipartition, when moving onto the point in the binary search where the ε-intervals on the grid have been identified for the locations of the left, middle, and right cuts, we can run the binary search for the location of the left cuts and, as this location is dependent on the locations of the middle and right cuts, this is sufficient to find the equipartition division. In running this binary search, however, if the investigated left cut does not have an associated middle cut, for instance, that would result in slices 1 and 2 of equal value based on the starting middle cut bounds, we need to find a way to update the left cut without finishing the protocol or introducing eval queries other than the cached queries. The method settled on leverages the monotonicity and hungriness of the valuation functions. Suppose a middle cut does not exist that would result in equal-valued slices 1 and 2. In that case, that must mean that all middle cut locations within the ε-interval would result in a slice 2 of lower value or higher value than slice 1 given the investigated left cut position. This is because the value of slice 2 must strictly increase as the location of the middle cut moves to the right, and if there exists some middle cut location where slice 2 has a lower value than slice 1 and some other location where slice 2 has a higher value than slice 1, by intermediate value theory, there must exist a middle cut location that would give equal-valued slices 1 and 2. Because of this, if there is no middle cut that would result in equal-valued slices 1 and 2 for the investigated left cut, we can find the value of slice 2 for the middle cut at the midpoint of the ε-interval and, if it is less than the value of slice 1, the left cut must be moved left; otherwise, it must be moved right so we can update the left cut bounds accordingly in the binary search. This method is used analogously for other binary searches of eval queries that we do not solve algebraically

once ε-intervals are identified.

4.4.3 Runtime Tests

Improvements	Runtime (secs)
None	73.835
Eval Queries	38.708
Binary Search	8.160
Eval Queries and Binary Search	4.531

Table 4.4: Runtime Tests for Hollender-Rubinstein Improvements

These runtime tests were run offline on my personal computer [1], so they do not include the time taken for the frontend to communicate with the backend. The tests use a cake size of 5, which was deemed a reasonable approximation of the cake size inputted by platform users. The runtimes are averages of 50 different runs of versions of the Hollender-Rubinstein algorithm, which include different combinations of the improvements. Further information on the methodology for the runtime tests not discussed here can be found in Chapter 6.

From table (4.4), it is clear that these two improvements substantially impacted the algorithm's runtime. The less significant of the two changes, more efficient eval queries, results in a runtime nearly half that of the initial implementation. The change to how binary searches are carried out gives a much larger runtime reduction by almost an order of magnitude. Together, they lead to an algorithm that runs in just short of 5 seconds as opposed to the initial runtime of around 74 seconds. This huge change illustrates the necessity of these changes. Even with just the new eval query implementation, the algorithm would still take far too long to reasonably expect users to wait for a division, which would harm user retention of the platform. The more involved improvements to how binary searches are handled are what make the algorithm feasible to include on the platform. Based on this large reduction in runtime from the way binary searches are implemented, it is not unreasonable to assume that implementing further algebraic methods in future could make it so runtime is no longer a concern for the Hollender-Rubinstein algorithm as it exists on Fair Slice.

[1] ASUS VivoBook 14, intel Core i7 10th gen processor

Chapter 5

The Piecewise-Constant Algorithm

To my knowledge, the next algorithm implemented did not have substantial research behind it. As such, it will be referred to descriptively as ”piecewise-constant” rather than by the names of the researchers who formalised its implementation. This algorithm works for an unlimited number of agents with the prerequisite that their valuation functions be piecewise-constant. Interestingly, this algorithm works by splitting agents' valuation functions into segments wherein all segments have constant value for all agents and does not split it into an ε-grid to aid approximation. Because of this, no modifications are made to eval queries or cut queries to leverage this construction, as eval queries will essentially be a series of linear multiples of segment values and sizes.

5.1 Preprocessing of Valuation Functions

Before beginning the algorithm procedure, the valuation functions must first be broken into segments, wherein each segment is a portion of the cake where each agent has a constant value at each point within this portion (i.e. a segment cannot comprise the portion of the cake between 0.4 and 0.6 if an agent's valuation function returns a different value at 0.45 than it does at 0.55). By this definition, a segment can technically be a portion of the cake wherein all agents have zero value for the cake. While the algorithm would still work if these segments were included, it would not aid its ability to find an envy-free division as varying the location of a cut within such a segment would not affect agent valuations and would unnecessarily harm runtime, so they are removed from the set of segments.

The segments are identified simply by finding a set of all start and endpoints of each piecewise-constant section for all agents, not including duplicated points. This gives a

grid of points in $[0,1]$ that make the initial set of segment start and endpoints. Then, the value of each segment is found for each agent by referencing its position relative to the agent's initial valuation function. The area of each segment is also found for each agent so it can be referenced by the algorithm later and does not need to be calculated every time to save on runtime. The segments are then investigated and removed from the set of segments if and only if all agents have zero value for this segment.

5.2 Implementation of The Algorithm

We will discuss the algorithm's implementation for the three-agent case, as it can easily be applied to cases with more agents with minor modifications. The idea is to iterate over all combinations of two segments and investigate them as possible positions for cuts to satisfy an envy-free division. This is done by iterating over the permutations of agents, such that the permutation $\{2,3,1\}$ would mean agent 2 would get slice 1, agent 2 would get slice 3, and agent 1 would get slice 3, and finding the linear functions of each slice value with the inputs being the cut position within the investigated segments.

These linear functions all have similar constructions in that they comprise a constant component, being the sum of areas of uninvestigated segments, and a variable component, being a multiple of one or more investigated segment's value by the amount of the segment that is included in the slice as dictated by the cut position. The linear functions for the values of the first and last slice are univariate, as either the start or endpoint of the slice will be the edge of the cake and, therefore, not variable. To represent these functions mathematically, we will describe segments as $S_i \mid i \in \{1,\ldots,n\}$ where n is the number of segments. The investigated segments for cut one and two positions will be written as S_{C_1} and S_{C_2}, respectively. The characteristics of the segments, value, startpoint, and endpoint will then be written as $S_{i,value}$, $S_{i,start}$, and $S_{i,end}$ respectively.

From this, the linear functions for agent values are as follows:

- Slice 1, where x is the position of cut one:

$$\sum_{i=1}^{C_1-1} S_{i,value} \cdot (S_{i,end} - S_{i,start}) + S_{C_1,value} \cdot (x - S_{C_1,start})$$

- Slice 2, where x is the position of cut one and y is the position of the cut two:

$$\sum_{i=C_1+1}^{C_2-1} S_{i,value} \cdot (S_{i,end} - S_{i,start}) + S_{C_1,value} \cdot (S_{C_1,end} - x) + S_{C_2,value} \cdot (y - S_{C_2,start})$$

- Slice 3, where y is the position of the cut two:

$$\sum_{i=C_2+1}^{n} S_{i,value} \cdot (S_{i,end} - S_{i,start}) + S_{C_2,value} \cdot (S_{C_2,end} - y)$$

While the investigated segments technically have variable value for each agent with respect to the cut position therein and all other segment values have constant value, it is important to note that all segment quantities $S_{i,value}$, $S_{i,start}$, and $S_{i,end}$ are treated as constants within the code. The only variables within these functions are x and y, which represent the cut positions within the investigated slices.

Once these linear functions are identified, this can be considered an optimisation problem where the objective function is 0 (The objective is 0 because we do not aim for optimal divisions, just envy-free. If the envy-freeness constraints are satisfied, the problem is solved). The constraints are constructed according to the permutation of agents we are investigating. For instance, if the permutation is $\{2, 1, 3\}$ we need to add constraints that agent 2's value for slice 1 is greater than slice 2 and 3, agent 1's value for slice 2 is greater than slice 1 and 3, and agent 3's value for slice 3 is greater than slice 1 and 2. We must also add a constraint that cut one is less than cut two. This is automatically satisfied in most cases as the bounds of these cuts are the start and endpoints of the investigated segments, and the investigated segment for cut one is never after the investigated segment for cut two, but in the case where both segments are the same, the constraint is necessary. With these constraints, the scipy solver for optimisation [13] will either return an envy-free division or return that no such division exists, at which point the loop will continue.

A similar implementation is used for four agents. Below is an example of the four agent Piecewise-Constant steps as they are returned on Fair Slice.

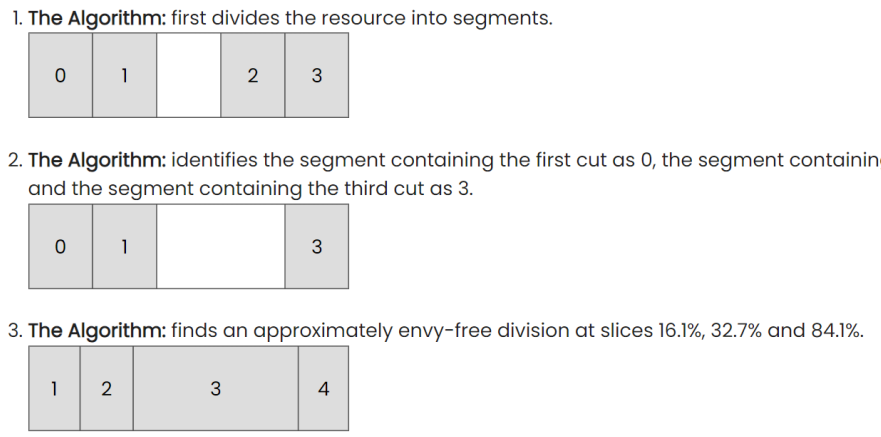

Figure 5.1: four agent Piecewise-Constant steps

5.3 Future Work for this Algorithm in Fair Slice

A possible improvement for this algorithm is the optimisation method. The algorithm currently uses a least squares approach, but the problem can be considered a linear programming problem and, therefore, can be solved via LP methods. It is tough to know ahead of time if this would result in a runtime improvement, but I believe it would be an interesting area to look into for further iterations of the Fair Slice platform.

The current scipy optimisation method [13] utilised in the implementation of this algorithm has a default tolerance that is not explicitly referenced in the documentation. Another possible area of improvement would be to investigate many different tolerances for the solver and log the associated runtime for each before deciding on the optimal tradeoff between runtime and accuracy.

A problem with the current implementation of this algorithm in the Fair Slice platform is a result of the agent's valuation functions not being read by the platform until after an algorithm is selected. In most cases, this is not a problem. The platform does consider the number of agents and only allows algorithms that work for that number of agents. But, for the unique prerequisite of piecewise-constant valuations, the platform cannot know if the Piecewise-Constant algorithm is suitable. This means it will be an option in many cases where it will only return an error if selected, which could be frustrating for users. This issue is mitigated by the interactive course for

Fair Slice, which describes the algorithm and its caveats at a high level. Still, this alone cannot eliminate the problem, as it is unreasonable to expect all users to read an interactive course before trying out the resource-splitting tool. Further work could be done on the Fair Slice platform to read valuation functions before suggesting algorithms and eliminate this problem. This further work could even be extended to consider more factors than just the form of the valuation functions, instead considering how the valuation functions inputted might affect the runtime of certain algorithms. This feature could then be leveraged to offer suggestions of algorithms based on runtime. This would be especially useful in future years when the repertoire of available algorithms will inevitably increase.

Another problem to consider is its complexity. Unlike the Brânzei-Nisan and Hollender-Rubinstein Algorithms, which have polylog complexity, the Piecewise-Constant algorithm has exponential complexity with respect to the agent number. This is a significant barrier to its feasibility as it can realistically only be run for low numbers of agents. A possible modification to this issue would be to preprocess the agent and segment combinations to disqualify any permutations that would violate proportionality, which is a prerequisite to envy-freeness. However, without prior testing, it is difficult to say with certainty that this preprocessing would not be overly complex in its implementation.

Chapter 6

Runtime Tests

6.1 Methodology

After implementing these three new algorithms on the Fair Slice platform, I ran several runtime tests for each newly implemented algorithm and the already implemented Selfridge-Conway algorithm for comparison. These tests were run on my personal computer [1]. They were run offline, so they do not include the time taken for the frontend to communicate with the backend, which is important to keep in mind when comparing Selfridge-Conway, which was written in the frontend, to the other three agent algorithms. The method I employed for these tests was first to set the cake size, that is, the number of sections in the piecewise-linear valuation functions, and then to generate uniform random valuations between 0 and 10 (the maximum allowed value on Fair Slice) for each agent and section of the cake. These valuation functions were then passed into the algorithm functions as inputs, and the runtime was saved to a .csv file. Because of the limitations inherent to the piecewise-constant algorithm, all valuation functions tested were piecewise-constant to ensure that the same valuation functions were tested for all new algorithms so that the compared runtimes were fair. Because the Selfridge-Conway algorithm was written in TypeScript, where the other algorithms were written in Python, it would not have been easy to use the same valuation functions generated in Python for tests done on Selfridge-Conway in TypeScript. Because the benefits of using the same valuation functions were outweighed by the difficulty inherent in the task, I opted to use different valuation functions in this case. I believe the averaging of runtimes should offset any issues this may cause, but it should still be taken into account. Each runtime returned for each algorithm and cake size averages 20 separate tests. This is

[1] ASUS VivoBook 14, intel Core i7 10th gen processor

to minimise the variance inherent in each algorithm. For instance, if Brânzei-Nisan or Hollender-Rubinstein find that the equipartition for Agent 1 is itself an envy-free division, that reduces the runtime of these algorithms considerably. Likewise, suppose the piecewise-constant algorithm finds that the segments 1 & 2 contain the cuts for an envy-free division. In that case, the runtime will be much smaller than expected on average. The investigated cake sizes began at 5, which was deemed a realistic cake size that a platform user would input. The cake sizes were then incremented by some amount that could capture the rate of runtime increase without necessitating a prohibitively large amount of tests. For the three agent algorithms, increasing cake sizes by a factor of 4 was deemed reasonable, whereas, for the four agent algorithms, a factor of 2 was chosen as the runtimes increased at a much faster rate. When one of the algorithms reached a prohibitively large runtime, final tests were run for the remaining algorithms for much larger cake sizes to see how long it would take for these algorithms to reach comparable runtimes.

The large number of tests carried out had a dual purpose of identifying bugs in the code that occurred in rare enough circumstances that they could not be identified by hand. These bugs were primarily a result of floating point division, leading to unforeseen behaviours in the code, and were solved by widespread additions of tolerances in inequalities and equalities throughout the code.

6.2 Results and Conclusions

As seen in table (6.1), beginning with the three agent algorithms and a cake size of 5, there is a clear and sizable difference in the runtimes of the three algorithms with Brânzei-Nisan taking more than an order of magnitude as much time as Selfridge-Conway and Piecewise-Constant taking more than two orders of magnitude as much time. This illustrates the large difference in complexity between the frontend and backend algorithms. This difference in complexity can be best understood by comparing how each algorithm finds cut positions. As previously discussed, Brânzei-Nisan utilises a binary search to find the exact cut positions at each iteration of the algorithm's main loop [6], and the Piecewise-Constant algorithm inspects each combination of segments and solves least squares optimisation problems until it finds a combination of segments that leads to a division that satisfies envy-freeness. Brânzei-Nisan was implemented in this way to conform to the implementation as laid out in the literature, but the piecewise-linear form of the agent's valuation functions as inputted in the Fair Slice platform

Agent Number	Cake Size	Algorithm	Runtime (secs)
Three Agents	5	Selfridge-Conway	0.00156
		Brânzei-Nisan	0.0429
		Piecewise-Constant	0.360
	20	Selfridge-Conway	0.00133
		Brânzei-Nisan	0.0363
		Piecewise-Constant	3.843
	80	Selfridge-Conway	0.00148
		Brânzei-Nisan	0.0706
		Piecewise-Constant	82.476
	320	Selfridge-Conway	0.00199
		Brânzei-Nisan	0.169
		Piecewise-Constant	1405.514
	320000	Selfridge-Conway	0.609
		Brânzei-Nisan	125.849
Four Agents	5	Hollender-Rubinstein	4.931
		Piecewise-Constant	6.431
	10	Hollender-Rubinstein	7.915
		Piecewise-Constant	30.369
	20	Hollender-Rubinstein	8.080
		Piecewise-Constant	277.139
	40	Hollender-Rubinstein	11.123
		Piecewise-Constant	1989.244
	6000	Hollender-Rubinstein	1331.445

Table 6.1: Runtime Tests for Fair Slice Algorithms

yields a more time-efficient, if less dynamic, method for identifying cut positions. This method involves taking in the start cut position and expected value of the slice as inputs and then iterating over the sections of the piecewise-linear valuation function beginning at the start cut position, adding the values of each section until a point is reached where a section is identified where adding its value would yield a value greater than the expected value inputted [8]. At this point, the exact cut position can be found by interpolation in this section. While this is not how cut queries are carried out in the cake-cutting literature, this simpler frontend algorithm implementation is much quicker than the alternatives as it necessitates the same computation as a standard eval query. As the cake sizes are increased, we can see how the runtimes evolve. For Selfridge-Conway, the increments in cake size are not large enough to see a notable computation increase that variability can't explain. Brânzei-Nisan shows a clearer trend due to the binary search involving enough value queries for the effect of more sections to iterate over to be more easily observed. The clearest trend is seen in the Piecewise-Constant algorithm due to its quadratic growth in the worst-case amount of segment combinations to expect. This quadratic factor comes from the fact that the number of segment combinations increases according to the formula for triangular numbers; in essence,

$$\begin{aligned} |\,S\,| &= \sum_{i=1}^{n} i && \text{where } n \text{ is the cake size} \\ &= n(n+1)/2. \end{aligned}$$

This quadratic growth means that the worst-case amount of inspected segments for a cake size of 5 is $5(5+1)/2 = 15$, but grows to $320(320+1)/2 = 51,360$ for a cake size of 320. At this point, the average runtime of the Piecewise-Constant is already over 23 minutes, whereas the other algorithms still take much less than a second. To further illustrate the comparative efficiency of these two algorithms, even increasing the cake size by a factor of 1000 results in Selfridge-Conway taking just over half a second and Brânzei-Nisan just over two minutes.

The runtimes of the four-agent algorithms increase at a much faster rate than the three-agent algorithms. Comparing Hollender-Rubinstein to Brânzei-Nisan, Brânzei-Nisan's conditions that must be checked are the case where at least two agents prefer the left slice and the case where at least two agents prefer the middle slice. We do not need to check the case where at least two agents prefer the right slice because of how the equipartition is constructed. We also only need to consider one such case for the entirety of the process because the only to exit a condition is by entering an envy-free division. Hollender Rubinstein has far more conditions that must be checked. Also,

they must all be considered for the entirety of the process as one condition can be exited into another. This means the amount of computation necessary for completing a run of Hollender-Rubinstein is far more than that necessary for completing a run of Brânzei-Nisan. Likewise, because of the addition of the third cut position to consider, the worst-case number of segments for the Piecewise-Constant algorithm to consider now grows cubically with cake size instead of just quadratically. This is because the number of segment combinations now increases according to the formula for tetrahedral number; in essence,

$$\begin{aligned} |\,S\,| &= \sum_{i=1}^{n}\sum_{j=1}^{i} j \qquad\qquad \text{where } n \text{ is the cake size} \\ &= n(n+1)(n+2)/6 \end{aligned}$$

This cubic growth means that the worst-case number of inspected segments for a cake size of 5 is $5(5+1)(5+2)/6 = 35$, which is larger than the equivalent worst-case of the three-agent case by a factor of $7/3$. The worst-case for a cake size of 40 is $40(40+1)(40+2)/6 = 11480$. While there are fewer combinations than the three-agent case for a cake size of 320, the larger runtime makes sense when you consider the extra agent combinations to investigate and the additional constraints that must be adhered to by the least squares optimisation solver.

While the runtimes for the four-agent algorithms are much closer for a cake size of 5 than they were for the three-agent algorithms, Piecewise-Constant is still the slowest. This difference in runtimes only gets more substantial as the cake sizes are increased. For a cake size of 40, Piecewise-Constant takes longer than Hollender-Rubinstein by more than two orders of magnitude. To get a sense of how comparatively inefficient Piecewise-Constant is with respect to increasing cake sizes in the four-agent case, to get a comparable runtime to Piecewise-Constant at cake size 320 for Hollender-Rubinstein, it was necessary to run tests at cake size 6000.

It is important to note that while some algorithms are clearly faster than others, that doesn't mean they slower algorithms should be disincentivised to the user. The purpose of the Fair Slice platform is education. The nuances that make some algorithms faster or slower than others are exactly the nuances that make them interesting to discuss and learn about. While there is no situation discussed here where Piecewise-Constant is the most efficient algorithm, that does not imply that it was not worth implementing, as the goal of the platform is not simply to return an envy-free division but also to offer insights into methods employed and researched by those in the field of algorithms.

Chapter 7

Envy-Free Moving Knife Algorithm Idea for Five Agents

Condition A: Agent 1 is indifferent between its four favourite slices, and the remaining piece is (weakly) preferred by at least two of the four remaining agents.

Condition B: Agent 1 is indifferent between its three favourite slices, and the remaining two slices are each (weakly) preferred by at least two of the four remaining agents, with one of these agents being indifferent between the two.

These first two conditions are analogous to those described by Hollender-Rubinstein [10]

Condition C: Agent 1 is indifferent between its three favourite slices, and the remaining two slices are each (weakly) preferred by at least two of the four remaining agents, with one of these agents being indifferent between one of Agent 1's favourite slices and one of the remaining two slices.

Condition D: Agent 1 is indifferent between its two favourite slices, and the remaining three slices are each (weakly) preferred by at least two of the four remaining agents, with two of these agents being indifferent between one of the remaining slices not preferred by the other and a mutual remaining slice.

Condition E: Agent 1 is indifferent between its two favourite slices, and the remaining three slices are each (weakly) preferred by at least two of the four remaining agents, with one of these agents being indifferent between the three.

7.1 The algorithm

Begin by having Agent 1 split the cake into fifths according to their valuations. If this division is envy-free, return it. If not, begin at condition A (Note that if the division is not envy-free, condition A must be true) and increase the size of Agent 1's four favourite slices such that Agent 1 remains indifferent between them. Continue with this until condition A can no longer be true if the process is continued. At this point, continuing the trail from another condition must be possible, or an envy-free division must exist. For every condition, if continuing the trail for this condition by increasing the size of Agent 1's preferred slices is no longer possible, another trail can be pivoted to, or an envy-free division must exist. A proof for this last claim is offered in the appendix.

7.2 Discussion

As I have described it, the algorithm is a moving-knife algorithm based on the Hollender-Rubinstein algorithm [10]. It was conceived by considering existing moving-knife algorithms for multiple agent numbers. Many of the main algorithms discussed in this dissertation can be seen as extensions of one another. The cut-and-choose algorithm and specifically the moving-knife analogue proposed by Austin [1] can be considered as just finding the equipartition of the cake. The Barbanell-Brams [2] (or Brânzei-Nisan [10]) algorithm can be seen as an extension of Austin's procedure for three agents by adding a condition for the case where the equipartition is not envy-free. Hollender-Rubinstein then further extends this for four agents by adding a second condition. None of these extensions change anything about the algorithm they are making extensions of apart from recontextualising the conditions for an extra agent. Because of this, I started thinking about an equivalent extension for five agents where the equipartition step and condition A and B are analogous to the Hollender-Rubinstein algorithm, as indeed they are here, but for five agents. The next step was to find further conditions that cover all exit criteria for the existing conditions without introducing exit criteria that existing conditions or an envy-free division do not cover. I believe I have shown that conditions A through E satisfy this, but it is important to note that the proof of correctness of this fact discussed in the appendix is not a proof of correctness of the algorithm as a whole. It is a good first step as it shows that the invariant should in theory only be exited at an envy-free division but, without exhaustively checking each combination of slices for the exit cases discussed and applying monotonicity and hungriness to show that the

algorithm can continue and not get stuck, the algorithm cannot be assumed correct. At this point, this is simply an idea for an envy-free algorithm. Proving its correctness would take more time than I have available, but I think it would be an interesting area for further research. If it is correct, I think it would be worthwhile to work on discretising it so that computer programs can implement it. I think that the algorithm could be implemented similarly to how the Hollender-Rubinstein algorithm is implemented via nested binary searches, but it is impossible to know for certain without further research on the subject. If the algorithm is not correct, I would be surprised if the approach of generalising the Hollender-Rubinstein algorithm for more agents by adding conditions would not yield a correct algorithm in future. I am interested in seeing what it looks like when it is discovered.

Chapter 8

Conclusion

8.1 Summary

The initial aim of this project was to implement the Hollender-Rubinstein algorithm on the Fair Slice platform, adding a four-agent algorithm to the two and three-agent algorithms already implemented. After the background reading and seeing the similarities between the Brânzei-Nisan and Hollender-Rubinstein algorithms, this was expanded to include the implementation of the Brânzei-Nisan algorithm as a stepping stone toward the more complex Hollender-Rubinstein algorithm. After these goals were completed earlier than expected, the aims were again expanded to include the Piecewise-Constant algorithm, which did not have prior research behind it. Hence, it involved much more freedom in the implementation than the previous algorithms, which had more strict implementation descriptions in their associated research papers.

With the completed implementation of these algorithms, the Fair Slice platform has taken another step towards being the foremost resource for cake-cutting education online. Before I began work on the platform, while it was aesthetically pleasing, engaging, and easy to use, the repertoire of cake-cutting algorithms was relegated to two simple algorithms that are the most well-known to the general public. The addition of these more complex, less well-known algorithms offers users a better understanding of the field of cake-cutting as it exists today and might spark enthusiasm in many users to learn more and perhaps pursue an interest in the study of the greater field of computer algorithms. As the repertoire of cake-cutting algorithms grows with the Fair Slice platform, I am optimistic that this will only become more true.

A secondary objective of mine for this project was to attempt to develop an idea for a cake-cutting algorithm for five agents. This was undertaken by making modifications

to the existing four-agent algorithm discovered by Hollender-Rubinstein. While the algorithm is currently in its early stages of development and does not have a complete proof, I am hopeful that this or some similar algorithm will become a fixture in the field of envy-free cake-cutting. If it proves correct, I am hopeful it can be discretised similarly to the Hollender-Rubinstein algorithm and be the first efficient five-agent cake-cutting algorithm.

8.2 Further Work

- **Add more algorithms.** The most natural avenue for further work is to expand the Fair Slice platform by adding more algorithms. The platform was created so that expansion is straightforward, and I believe it should be home to as many different and interesting algorithms as possible.

- **Improve existing algorithms.** The algorithm with the clearest avenues for improvement is the Piecewise-Constant algorithm, and those improvements are laid out in its respective chapter. There are possible improvements to the other algorithms, though. One such improvement would be to leverage the way valuation functions are inputted. Because the platform has access to all the information of the valuation functions, the standard implementation of cut queries as binary searches of value queries isn't strictly necessary. Suppose runtime is deemed a priority over strict adherence to the algorithm implementations as they are described in their respective papers. In that case, I believe the runtimes of the Hollender-Rubinstein and Brânzei-Nisan algorithms could be substantially improved. Even if the choice is made to preserve the integrity of the algorithms' accurate implementations, the Hollender-Rubinstein algorithm could be further improved by implementing further algebraic methods described in its respective chapter.

- **Formalizing and discretizing the five-agent algorithm idea.** The final area for further work is to formalize a proof for the five-agent algorithm idea and, if it proves correct, modify it to be used by a computer. This would be a worthwhile result in its own right and could further tie into the project by implementing it in the Fair Slice platform in years to come if the algorithm can be implemented efficiently.

Bibliography

[1] A.K. Austin. Sharing a cake. *The Mathematical Gazette*, 66(437):212–215, 1982.

[2] Julius B. Barbanel and Steven J. Brams. Cake division with minimal cuts: envy-free procedures for three persons, four persons, and beyond. *Mathematical Social Sciences*, 48(3):251–269, 2004.

[3] Steven J. Brams and Alan D. Taylor. An envy-free cake division protocol. *The American Mathematical Monthly*, 102(1):9–18, 1995.

[4] Steven J. Brams and Alan D. Taylor. *Fair Division From Cake-Cutting to Dispute Resolution*. Cambridge University Press, 1996.

[5] Simina Brânzei and Noam Nisan. Communication complexity of cake cutting. *CoRR*, abs/1709.09876, 2017.

[6] Simina Brânzei and Noam Nisan. The query complexity of cake cutting. *ArXiv*, abs/1705.02946, 2017.

[7] Andy C. Ernst. https://fairslice.netlify.app/.

[8] Andy C. Ernst. https://github.com/andycernst/cake, 2023.

[9] George Gamow and Marvin Stern. *Puzzle Math*. Viking Press, 1958.

[10] Alexandros Hollender and Aviad Rubinstein. Envy-free cake-cutting for four agents, 2023.

[11] Ariel D. Procaccia. Cake cutting: not just child's play. *Commun. ACM*, 56(7):78–87, jul 2013.

[12] scipy. https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.linear$_s um_a ssignment.ht$

[13] scipy. https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html.

[14] Hugo Steinhaus. The problem of fair division. *Econometrica*, 16(1):101–104, jan 1948.

[15] Walter Stromquist. How to cut a cake fairly. *The American Mathematical Monthly*, 87(8):640–644, 1980.

[16] Gerhard Woeginger and Jiří Sgall. On the complexity of cake cutting. *Discrete Optimization*, 4:213–220, 06 2007.

Appendix A

Proof of Correctness the Validity of the Chosen Invariant

First, we will state the formal definitions of the conditions in the invariant.

Condition A: There exists a slice $k \in \{1, 2, 3, 4, 5\}$ such (i) that for all slices $t, t' \in \{1, 2, 3, 4, 5\} \setminus \{k\}$ such that $v_1(t) = v_1(t') \geq v_1(k)$, and (ii) there exists two agents $i, i' \in \{2,3,4,5\}$ such that $v_i(k) \geq \max_t v_i(t)$ and $v_{i'}(k) \geq \max_t v_{i'}(t)$.

Condition B: There exists two slices $k, k' \in \{1, 2, 3, 4, 5\}$ such (i) that for all slices $t, t' \in \{1, 2, 3, 4, 5\} \setminus \{k, k'\}$ such that $v_1(t) = v_1(t') \geq \max\{v_1(k), v_1(k')\}$, and (ii) for each slice $h \in \{k, k'\}$ there exists two agents $i, i' \in \{2,3,4,5\}$ such that $v_i(h) \geq \max_t v_i(t)$ and $v_{i'}(h) \geq \max_t v_{i'}(t)$ with (iii) one of these agents satisfying $v_i(k) = v_i(k') \geq \max_t v_i(t)$.

Condition C: There exists two slices $k, k' \in \{1, 2, 3, 4, 5\}$ such (i) that for all slices $t, t' \in \{1, 2, 3, 4, 5\} \setminus \{k, k'\}$ such that $v_1(t) = v_1(t') \geq \max\{v_1(k), v_1(k')\}$, and (ii) for each slice $h \in \{k, k'\}$ there exists two agents $i, i' \in \{2,3,4,5\}$ such that $v_i(h) \geq \max_t v_i(t)$ and $v_{i'}(h) \geq \max_t v_{i'}(t)$ with (iii) one of these agents satisfying $v_i(k) = v_i(t) \geq \max_{t'} v_i(t')$ for some slice $t \in \{1, 2, 3, 4, 5\} \setminus \{k, k'\}$.

Condition D: There exists three slices $k, k', k'' \in \{1, 2, 3, 4, 5\}$ such (i) that for all slices $t, t' \in \{1, 2, 3, 4, 5\} \setminus \{k, k', k''\}$ such that $v_1(t) = v_1(t') \geq \max\{v_1(k), v_1(k'), v_1(k'')\}$, and (ii) for each slice $h \in \{k, k', k''\}$ there exists two agents $i, i' \in \{2,3,4,5\}$ such that $v_i(h) \geq \max_t v_i(t)$ and $v_{i'}(h) \geq \max_t v_{i'}(t)$ with (iii) two of these agents satisfying $v_i(k) = v_i(k') \geq \max_t v_i(t)$ and $v_{i'}(k'') = v_{i'}(k') \geq \max_t v_i(t)$.

Condition E: There exists three slices $k, k', k'' \in \{1, 2, 3, 4, 5\}$ such (i) that for all slices $t, t' \in \{1, 2, 3, 4, 5\} \setminus \{k, k', k''\}$ such that $v_1(t) = v_1(t') \geq \max\{v_1(k), v_1(k'), v_1(k'')\}$, and (ii) for each slice $h \in \{k, k', k''\}$ there exists two agents $i, i' \in \{2,3,4,5\}$ such

that $v_i(h) \geq \max_t v_i(t)$ and $v_{i'}(h) \geq \max_t v_{i'}(t)$ with (iii) one of these agents satisfying $v_i(k) = v_i(k') = v_i(k'') \geq \max_t v_i(t)$.

Condition A proof: If we reach the point of increasing the value of Agent 1's favourite slices (we will call this a Trail A) where condition A will no longer hold, it must be because one of the two remaining agents 2 & 3, say Agent 3 wlog, will no longer prefer the remaining slice, k, and will instead prefer another slice k' if we continue along this trail (that is, Agent 3 is currently indifferent between k and k'. There are a few cases of this:

- Agents 4 & 5 prefer different slices $t, t' \notin \{k\}$ and Agent 3 prefers some slice $k' \notin \{t, t', k\}$, meaning this division is envy-free and the algorithm terminates
- Agents 4 & 5 prefer different slices $t, t' \notin \{k\}$ and Agent 3 prefers some slice $k' \in \{t, t'\}$, meaning this division satisfies condition B with agent 3 being the indifferent agent and we can continue along Trail B.
- Agents 4 & 5 prefer the same slice $t \notin \{k\}$ and Agent 3 prefers some slice $k' = t$, meaning this division again satisfies condition B with agent 3 being the indifferent agent and we can continue along Trail B.
- Agents 4 & 5 prefer the same slice $t \notin \{k\}$ and Agent 3 prefers some slice $k' \neq t$, meaning this division satisfies condition C with agent 3 being the indifferent agent and we can continue along Trail C.

Condition B proof: If we reach the point on Trail B, we must have reached a point where condition B will no longer hold. This is because either (i) Agent 1 is currently indifferent between its three favourite slices and one of the remaining slices, (ii) the indifferent Agent, say Agent 5 wlog, is now indifferent between the two remaining slice k and k' as well as a third slice $k'' \notin \{k, k'\}$, or (iii) one of the remaining agents is currently indifferent between one of the remaining slices k and some other slice $t \neq k$.

In case (i), Agent 1 is indifferent between its four favourite slices, and at least two agents prefer the remaining slice. This aligns with condition A, and we can continue along trail A.

In case (ii), there are the following scenarios to consider:

- Agents 2 & 3 prefer the same slice k and Agent 4 prefers slice k'. In this case, Agent 1 is indifferent between their three favourite slices and the two remaining slices are each preferred by two agents, one of which is indifferent between one

of these slices and one of Agent 1's favourite slices. This is in line with condition C, and we can continue along Trail C with agent 5 being held indifferent. An important note here is that, when continuing along trail C, Agent 5 can only be indifferent between some slice $k'' \notin \{k, k'\}$ and one of the slices k, k' as the other slice must monotonically decrease in value as the other four increase in value.

- Agents 2 prefers slice k, Agent 3 prefers slice k', and Agent 4 prefers some slice $t \notin \{k, k', k''\}$. This is an envy-free division, and the algorithm terminates.

- Agents 2 prefers slice k, Agent 3 prefers slice k', and Agent 4 prefers some slice k''. In this case, Agent 1 is indifferent between its two favourite slices and the remaining three slices are each preferred by at least two agents, one of which is indifferent between the three slices. This satisfies Condition E, and we can continue along Trail E with Agent 5 held indifferent.

In case (iii), there are the following scenarios to consider:

- Agents 2 & 3 prefer the same slice k' and Agent 4 is now indifferent between slices k and k'. In this case, we can pivot to a different trail B with Agent 4 held indifferent. This is analogous to the case where Agent 2 prefers slice k, Agent 3 prefers slice k', and Agent 4 is indifferent between slices k and k' and the case where Agent 3 prefers slice k', Agent 4 is indifferent between slices k and k' with Agent 1 preferring some slice $t \notin \{k, k'\}$

- Agents 2 & 3 prefer the same slice k' and Agent 4 is now indifferent between slices k and some slice $t \notin \{k, k'\}$. In this case, Agent 1 is indifferent between their three favourite slices and the two remaining slices are each preferred by two agents, one of which is indifferent between one of these slices and one of Agent 1's favourite slices. This satisfies Condition C, and we can continue along Trail C with Agent 4 held indifferent

- Agent 3 prefers slice k' and Agent 4 is now indifferent between slices k and some slice $t \notin \{k, k'\}$, with Agent 2 also preferring slice t. In this case, Agent 1 is indifferent between its two favourite slices, with the three remaining slices each preferred by at least two of the four remaining agents, with two of these agents being indifferent between one of the remaining slices not preferred by the other and a mutual remaining slice. This is in line with condition D, and we can continue along Trail D with Agents 4 & 5 being held indifferent.

- Agent 3 prefers slice k' and Agent 4 is now indifferent between slices k and some slice $t \notin \{k, k'\}$, with Agent 2 preferring some slice $t' \notin \{k, k', t\}$. This is an envy-free division, and the algorithm terminates.

Condition C proof: If we reach the point on Trail C, we must have reached a point where condition C will no longer hold. This is because either (i) Agent 1 is currently indifferent between its three favourite slices and one of the remaining slices, (ii) the agent that prefers the remaining slice k along with the indifferent agent is now indifferent between k and some slice $t \notin \{k,k'\}$ (note that they cannot prefer slice k' as it is monotonically decreasing in value), (iii) one of the agents that prefers the remaining slice k' not preferred by the indifferent agent is now indifferent between k' and some slice $t \neq k'$, or (iv) the indifferent agent is now indifferent between their initially preferred slices k and t as well as some slice $t' \notin \{k, k't\}$.

In case (i), Agent 1 is indifferent between its four favourite slices and the remaining piece is preferred by at least two of the four remaining agents. This is in line with condition A, and we can continue along Trail A. Note that Agent 1 cannot be indifferent between its three favourite slices, and the remaining slice not preferred by the indifferent agent k' because its value is monotonically decreasing.

In case (ii), the agent, say Agent 4, that prefers the remaining slice k along with the indifferent agent, say Agent 5, is now indifferent between k and some slice $t \notin \{k,k'\}$ is now a valid choice for the indifferent agent if we were to change to a different trail C. This is because, continuing along this new trail C with Agent 4 held indifferent, Agent 5 will no longer weakly prefer one of Agent 1's favourite slices. The fact that continuing along Trail C with Agent 5 held indifferent would result in condition C no longer being true means that, instead, holding Agent 4 indifferent would result in the size (and therefore value, by monotonicity and hungriness) of the slice k increasing quicker than if Agent 5 was held indifferent. This means that Agent 5 will strictly prefer slice k in this new trail C, and condition C will continue to be satisfied.

In case (iii), there are the following scenarios to consider:

- Given Agent 3 prefers slice k', Agent 4 prefers slice k, Agent 5 is indifferent between k and some slice $t \notin \{k, k'\}$, and Agent 2 is now indifferent between slices k' and k, this now satisfies condition B, so we can continue on trail B with agent 2 held indifferent.

- Given Agent 3 prefers slice k', Agent 4 prefers slice k, Agent 5 is indifferent between k and some slice $t \notin \{k, k'\}$, and Agent 2 is now indifferent between

slices k' and t, Agent 1 is indifferent between its two favourite slices, and the remaining three slices are each (weakly) preferred by at least two of the four remaining agents, with two of these agents being indifferent between one of the remaining slices not preferred by the other and a mutual remaining slice. This satisfies condition D, so we can continue along trail D with Agents 2 & 5 held indifferent.

- Given Agent 3 prefers slice k', Agent 4 prefers slice k, Agent 5 is indifferent between k and some slice $t \notin \{k, k'\}$, and Agent 2 is now indifferent between slices k' and some slice $t' \notin \{k, k', t\}$, this is an envy-free division and the algorithm terminates.

In case (iv), the indifferent agent, say Agent 5, is now indifferent between their initially preferred slices k, along with Agent 4, and t, preferred by Agent 1 and a third slice t' also preferred by Agent 1. Agents 2 & 3 prefer slice k'. In this case, no other agents' preferences have changed and this still satisfies Condition C, so we can pivot to a new trail C with Agent 5 now held indifferent between k and t'.

Condition D proof: If we reach the point on Trail D, we must have reached a point where condition D will no longer hold. This is because either (i) Agent 1 is currently indifferent between its two favourite slices and one of the remaining slices, (ii) one of the agents that is indifferent between slices k and k' also weakly prefers some slice $t \notin \{k, k'\}$, or (iii) one of the remaining agents is now indifferent between their preferred slice k and some slice $t \neq k$.

In case (i), there are the following scenarios to consider:

- Agent 1 is indifferent between its two preferred slices and the slice that the indifferent agents mutually prefer, say Agents 4 & 5 wlog, slice k'. In this case, Agent 1 is indifferent between its three favourite slices, and the remaining two slices are each (weakly) preferred by at least two of the four remaining agents, with one of these agents being indifferent between one of Agent 1's favourite slices and one of the remaining two slices. This satisfies condition C and we can continue along Trail C with either Agent 4 or 5 held indifferent, depending on which indifferent agent would result in the other strictly preferring slice k or k'' and not k'.

- Agent 1 is indifferent between its two preferred slices and one of the other slices, say slice k, preferred by Agents 3 & 4, with Agent 4 also weakly preferring

slice k'. The slice k'' is preferred by Agents 2 & 5 with Agent 5 also weakly preferring slice k'. In this case, Agent 1 is indifferent between its three favourite slices, and the remaining two slices are each (weakly) preferred by at least two of the four remaining agents, with one being indifferent between them. This satisfies condition B, and we can continue along trail B with Agent 5 being held indifferent.

In case (ii), there are the following scenarios to consider:

- Given Agent 3 prefers slice k, Agent 2 prefers slice k', Agent 4 is indifferent between slices k'' and k' and Agent 5 is now indifferent between slices k, k', and k'', this now satisfies condition E and we can continue along Trail E with Agent 5 held indifferent.

- Given Agent 3 prefers slice k, Agent 2 prefers slice k', Agent 4 is indifferent between slices k'' and k' and Agent 5 is now indifferent between slices k, k', and some slice $t \notin \{k, k'\}$, this is an envy-free division and the algorithm terminates.

In case (iii), there are the following scenarios to consider:

- Given Agent 3 prefers slice k, Agent 4 is indifferent between slices k'' and k' and Agent 5 is indifferent between slices k, k', and Agent 2 is now indifferent between slice k'' and slice k, this still satisfies condition D, and we can continue on a new trail D with Agents 2 and 4 held indifferent.

- Given Agent 3 prefers slice k, Agent 4 is indifferent between slices k'' and k' and Agent 5 is indifferent between slices k, k', and Agent 2 now is indifferent between slice k' and slice k'', this still satisfies condition D, and we can continue on a new trail D with Agents 2 and 5 held indifferent.

- Given Agent 3 prefers slice k, Agent 4 is indifferent between slices k'' and k' and Agent 5 is indifferent between slices k, k', and Agent 2 is now indifferent between slice k'' and slice $t \notin \{k, k', k''\}$, this is an envy-free division, and the algorithm terminates.

Condition E proof: If we reach the point on Trail E, we must have reached a point where condition E will no longer hold. This is because either (i) Agent 1 is currently indifferent between its two favourite slices and one of the remaining slices, (ii) The agent that is indifferent between slices k, k', and k'' also weakly prefers some slice

$t \notin \{k, k', k''\}$, or (iii) one of the remaining agents is now indifferent between their preferred slice k and some slice $t \neq k$.

For case (i), this division now satisfies condition B, and we can continue along trail B with the same agent held indifferent.

For case (ii), this is an envy-free division, and the algorithm terminates.

For case (iii), there are the following scenarios to consider:

- Given Agent 5 is indifferent between slices k, k', and k'', Agent 4 prefers slice k'', Agent 3 prefers slice k', and Agent 2 is now indifferent between slice k and k', this now satisfies condition D, and we can continue along trail D with Agents 2 and 5 held indifferent.

- Given Agent 5 is indifferent between slices k, k', and k'', Agent 4 prefers slice k'', Agent 3 prefers slice k', and Agent 2 is now indifferent between slice k and some slice $t \notin \{k, k', k''\}$, this is an envy-free division, and the algorithm terminates.