A Discrete and Bounded Envy-free Cake Cutting Protocol for Any Number of Agents

Haris Aziz
Data61, CSIRO and UNSW Australia
Sydney, Australia, NSW 2052
Email: haris.aziz@data61.csiro.au

Simon Mackenzie
Data61, CSIRO and UNSW Australia
Sydney, Australia, NSW 2052
Email: simon.mackenzie@data61.csiro.au

***Abstract*—We consider the well-studied cake cutting problem in which the goal is to find an envy-free allocation based on queries from n agents. The problem has received attention in computer science, mathematics, and economics. It has been a major open problem whether there exists a discrete and bounded envy-free protocol. We resolve the problem by proposing a discrete and bounded envy-free protocol for any number of agents. The maximum number of queries required by the protocol is $n^{n^{n^{n^{n^n}}}}$. Even if we do not run our protocol to completion, it can find in at most n^{n+1} queries an envy-free partial allocation of the cake in which each agent gets at least $1/n$ of the value of the whole cake.**

I. Introduction

The existence of a discrete and bounded envy-free cake cutting protocol has remained a major open problem for at least two decades. In this paper, we settle the problem by presenting a general discrete envy-free protocol for any number of agents. The protocol is for the cake cutting setting that is a versatile mathematical model for allocation of a heterogeneous divisible good among multiple agents with possibly different preferences over different parts of the cake. The main applications of cake cutting are fair scheduling, resource allocation, and conflict resolution [10]. Cake cutting has been extensively studied within computer science [18] and the social sciences [26]. Since various important divisible resources can be captured by cake cutting, the problem of fairly dividing the cake is a fundamental one within the area of fair division and multiagent resource allocation [7, 17, 20].

A cake is represented by an interval $[0, 1]$ and each of the n agents has a valuation function over pieces of the cake that specifies how much that agent values a particular subinterval. The main goal is to divide the cake fairly. Among various fairness concepts proposed by social scientists, a prominent one is *envy-freeness*. An allocation is envy-free if no agent would prefer to take another agent's allocation instead of his own. Although an envy-free allocation is guaranteed to exist even with $n - 1$ cuts [25][1], *finding* an envy-free allocation is a challenging problem which has been termed "one of the most important open problems in 20th century mathematics" by Garfunkel [13].

[1]Su [25] pointed out that the existence of an envy-free cake allocation can be shown via an interesting connection with Sperner's Lemma.

a) Motivation and Contribution: Since the valuations of agents over the cake can be complex, eliciting each agent's complete valuations function over the cake is computationally infeasible. A natural approach in cake cutting protocols is to query agents about their valuations of different portions of the cake and based on these queries propose an allocation. A cake cutting protocol is *envy-free* if each agent is guaranteed to be non-envious if he reports his real valuations. For the case of two agents, the problem has a well known solution in the form of the *Divide and Choose* protocol: one agent is asked to cut the cake into equally preferred pieces and the other agent is asked to choose the preferred piece. For the case of three agents, an elegant and bounded protocol was independently discovered by John L. Selfridge and John H. Conway around 1960 [Page 116, 7]. Since then, an efficient general envy-free protocol for any number of agents has eluded researchers.

In 1995, Brams and Taylor [6] made a breakthrough by presenting an envy-free protocol for *any* number of agents. Although the protocol is guaranteed to terminate in finite time, there is one drawback of the protocol: the running time or number of queries and even the number of cuts required is unbounded even for four agents. In other words, the number of queries required to identify an envy-free allocation can be arbitrarily large for certain valuations functions. Procaccia [18] terms unboundedness as a "serious flaw". Brams and Taylor mentioned the problem of proposing a bounded envy-free protocol even for $n = 4$. The problem has remained open and has been highlighted in several works [6, 7, 11, 22, 17, 18, 14, 20, 21]. Procaccia [17] calls it an interesting challenge within theoretical computer science. In recent work, Aziz and Mackenzie [1] proposed a bounded and discrete envy-free protocol for $n = 4$. Despite this progress, the general problem for any number of agents had remained unaddressed.

In this paper, *we present a discrete envy-free protocol for any number agents that requires a bounded number of queries and hence a bounded numbers of cuts of the cake* thereby closing the central open problem in the field of cake cutting. The bound for the number of queries is $n^{n^{n^{n^{n^n}}}}$. Previously, no discrete protocol was known that even uses bounded number of cuts. Apart from using some classic ideas in cake-cutting, our protocol requires some radically different techniques. Our result is surprising because leading experts in the area have conjectured that a bounded and discrete envy-free protocol does not exist [18].

We additionally show that even if we do not run our protocol to completion, it can find in at most n^{n+1} queries a partial allocation of the cake that achieves proportionality (with respect to the whole cake) and envy-freeness.

Apart from introducing new techniques, our protocol relies on some ideas from previous protocols. At the heart of our protocol is the idea of domination or 'irrevocable advantage' [6]. An agent i dominates another agent j if he is not envious of j even if the unallocated cake is given to j. Note that if agents in subset $S \subset N$ dominate all the agents in $N \setminus S$, then agents in S can simply let the agents in $N \setminus S$ worry about the remaining cake since they will not be envious if a single agent in $N \setminus S$ got all the remaining unallocated cake. Our protocol uses some high level ideas from our previous protocol for four agents: (i) We use a 'Core Protocol' over the unallocated cake (aka residue) with one agent as the specified cutter that results in an envy-free partial allocation and a smaller residue. The Core Protocol is called again and again on the latest residue so as to make the residue even smaller. As the new calls to the Core Protocol are made, the partial allocation remains envy-free. (ii) Every time agents make trims or evaluate pieces of cake, we keep track of information gathered. (iii) We make agents exchange some parts of their already allocated pieces to obtain new dominations. The exchange was referred to as a *permutation* in [1]. In some steps, our protocol also uses a simple fact: envy-freeness implies proportionality with respect to the allocated cake (i.e., each agent gets at least $1/n$ value of the allocated cake).

Throughout the overall protocol, we have cake that has been allocated and some cake that is the unallocated residue. During the protocol, we ensure that the cake that has been allocated to the agents has been done so in an envy-free manner. Overall, our envy-free protocol (*Main Protocol*) takes as input cake and a set of agents and returns an envy-free allocation of the whole cake. In order to obtain the envy-free allocation, the Main Protocol calls three other protocols. It repeatedly calls the *Core Protocol* a bounded number of times. The Core Protocol is the work-horse of the overall protocol that is called again and again to further allocate unallocated cake in any envy-free manner. The other protocols—GoLeft and Discrepancy are used to make one set of agents dominate the others so that the problem reduces to finding an envy-free allocation for fewer agents and the Main Protocol can be called again for this sub-problem.

After calling the Core Protocol many times, if there is still some unallocated cake and there is sufficient difference in the estimation of the value of a cake piece between some agents, then the Discrepancy Protocol is called by the Main Protocol. The goal of the Discrepancy Protocol is to either exploit difference in valuations or else to ensure that on a very rough scale, agents have similar valuations. If after calling the Discrepancy Protocol, the cake still has not been completely allocated, the GoLeft Protocol is called by the Main Protocol. The GoLeft Protocol is used to reallocate pieces of the cake among the agents after additionally adding some crumbs from the residue. The reallocation is useful to obtain new dominations until one set of agents dominate the remaining agents which means that we have reduced the problem to allocating the remainder of the cake among the dominated agents. The different protocols that we present are summarized in Figure 1. For complete details of the protocols, please see the accompanying technical report [2].

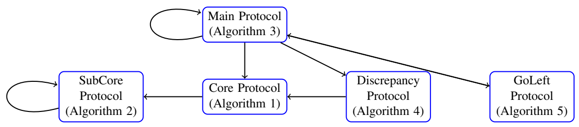

Main Protocol (Algorithm 3)
SubCore Protocol (Algorithm 2)
Core Protocol (Algorithm 1)
Discrepancy Protocol (Algorithm 4)
GoLeft Protocol (Algorithm 5)

Fig. 1: The envy-free protocol for n agents relies on various protocols. A protocol points to another protocol if it calls the latter. The Main Protocol is called to get an envy-free allocation.

b) New Techniques and Ideas: Despite some high level similarities with the approach of the recent protocol for four agents [1], our general protocol requires a number of new concepts and techniques. These concepts and techniques will be formalized and explained in the later sections but we mention them briefly and informally. We first rely on a notion of discrepancy that is based on the idea of how differently agents view a piece of cake. It is well known in fair division that the more different the agents' valuations are, the easier it is to achieve fairness. We use this high-level idea but make sure that discrepancy is *not* based on the absolute difference or ratio of the values of two agents for a cake piece but is based on the ratio of each agent's value of the piece of cake to that of their value for the remaining unallocated cake (called residue). This approach is one of the key ideas for obtaining a bounded protocol rather than simply a finite protocol.

Another key idea to achieve a bounded protocol is that we run the Core Protocol enough times to ensure there are a sufficient number of Core Protocol allocation snapshots (allocation outcomes of various calls of the Core Protocol) that are *isomorphic*.[2] We will keep track of these snapshots to later make more agents dominate each other.

Further operations such as exchanges of allocated pieces are then implemented on such isomorphic core snapshots by exploiting their structure. In order to achieve boundedness of the protocol, we also define and work with four carefully chosen bounds that are functions of the number of agents. The biggest of the four bounds is the maximal number of query operations required to complete the protocol.

We use a new technique called *extraction* whereby part of the residue is systematically trimmed off so that it can potentially be combined with some agent's allocation in the Core Protocol. When a particular extracted piece of cake is given to an agent in addition to his allocated piece in a single snapshot of the Core Protocol, we refer to this as *attachment*.

[2]The notion of isomorphic snapshots will be defined later.

Attachment is helpful to systematically enable more agents to exchange each others' pieces in snapshots of the Core Protocol without causing any agent to be envious. The exchange of pieces consequently allows for 'permutations' and helps agents dominate each other.

In order to methodically track the possible permutations/exchanges, we keep track of a *permutation graph* in which nodes correspond to agents and an agent i points to agent j if i is willing to replace his allocation in a subset of isomorphic snapshots of the Core Protocol with j's allocated pieces along with extracted pieces attached to j's allocated pieces. The tiny bit more cake attached to attract agent i can in principle cause other agents to be envious so we have to compensate the other agents elsewhere so that envy is not introduced. This is done in the Goleft protocol by discarding isomorphic snapshots from the set of working snapshots in which further exchanges will take place. The attachments and exchanges take place in the GoLeft Protocol. The interplay between the permutation graph and the working set of isomorphic core snapshots is technically one of the most interesting parts of the overall protocol and is the central part of the GoLeft Protocol.

c) Related Work: Cake cutting problems originated in the 1940's when famous mathematicians such as Banach, Knaster, and Steinhaus initiated serious mathematical work on the topic of fair division.[3] Since then, the theory of cake cutting algorithms has become a full-fledged field with at least three books written on the topic [4, 7, 20]. The central problem within cake cutting is finding an envy-free allocation [12].

When formulating efficient cake cutting protocols, a typical goal is to minimize the number of cuts while ignoring the number of valuations queried from the agents. In principle, the actual complexity of a problem or a protocol depends on the number of queries. When considering how efficient a protocol is, it is useful to have a formal query model for cake cutting protocols. Robertson and Webb [20] formalized a simple query model in which there are two kinds of queries: Evaluation and Cut. In an Evaluation query, an agent is asked how much he values a subinterval. In a Cut query, an agent is asked to identify an interval, with a fixed left endpoint, of a particular value. Although, the query model of Robertson and Webb [20] is very simple, it is general enough to capture all known protocols in the literature. Note that if the number of queries is bounded, it implies that the number of cuts is bounded in the Robertson and Webb model. The protocol that we present in this paper uses a bounded number of queries in the Robertson and Webb model.

There is not too much known about the existence of a bounded envy-free protocol for general n except that any envy-free cake cutting algorithm requires $\Omega(n^2)$ queries in the Robertson-Webb model [16, 18]. Also, for $n \geq 3$, there exists no finite envy-free cake cutting algorithm that outputs *contiguous* allocations in which agents get a connected piece with no gaps [24]. Brams et al. [8] and Barbanel and Brams [5] presented envy-free protocols for four agents that require 13 and 5 cuts respectively. However, the protocols are not only unbounded but also not finite since they are *continuous* protocols that require the notion of a *moving knife*. Apart from the unbounded Brams and Taylor envy-free protocol for n agents, there are other general envy-free protocols by Robertson and Webb [19] and Pikhurko [15] that are also unbounded. Aziz and Mackenzie [1] proposed a bounded and discrete envy-free protocol for $n = 4$. There are positive algorithmic results concerning envy-free cake cutting when agents have restricted valuations functions [3, 9, 10] or when some part of the cake is left unallocated [22, 21].

[3]Hugo Steinhaus presented the cake cutting problems to the mathematical and social science communities on Sep. 17, 1947, at a meeting of the Econometric Society in Washington, D.C. [20, 23].

II. Preliminaries

a) Model: We consider a cake which is represented by the interval $[0,1]$. A *piece of cake* is a finite union of disjoint subintervals of $[0,1]$. We will assume the standard assumptions in cake cutting. Each agent i in the set of agents $N = \{1,\dots,n\}$ has his own valuation function V_i that specified his values over subintervals of interval $[0,1]$. The valuations are (i) *defined on all finite unions of the intervals*; (ii) non-negative: $V_i(X) \geq 0$ for all $X \subseteq [0,1]$; (iii) *additive*: for all disjoint $X, X' \subseteq [0,1]$, $V_i(X \cup X') = V_i(X) + V_i(X')$; (iv) *divisible* i.e., for every $X \subseteq [0,1]$ and $0 \leq \lambda \leq 1$, there exists $X' \subseteq X$ with $V_i(X') = \lambda V_i(X)$.

In order to ascertain the complexity of a protocol, Robertson and Webb presented a computational framework in which agents are allowed to make two kinds of queries: (1) for given $x \in [0,1]$ and $r \in \mathbb{R}^+$, Cut query asks an agent to return a point $y \in [0,1]$ such that $V_i([x,y]) = r$ (2) for given $x, y \in [0,1]$, Evaluate query asks an agent to return a value $r \in \mathbb{R}^+$ such that $V_i([x,y]) = r$. A cake cutting protocol specifies how agents interact with queries and cuts. All well-known cake cutting protocols can be analyzed in terms of the number of queries required to return a fair allocation. A cake cutting protocol is *bounded* if the number of queries required to return a solution is bounded by a function of n irrespective of the valuations of the agents.

b) Terms and Conventions: We now define some terms and conventions that we will use in the paper. An allocation is *partial* if it does not necessarily allocate the whole cake. Given an envy-free partial allocation X of the cake and an unallocated residue R, we say that agent i *dominates* agent j if i does not become envious of j even if all of R were to be allocated to j: $V_i(X_i) \geq V_i(X_j) + V_i(R)$.

In the cake cutting protocols we will describe, an agent may be asked to trim a piece of cake so that its value equals the value of a less valuable piece. Agents will be asked to trim various pieces of the cake to make a given piece equal to the value of another piece. When an agent trims a piece of cake, he will trim it from the left side: the main piece (albeit trimmed) will be on the right side.

III. The Protocol

The overall protocol we present is essentially the Main Protocol that calls other protocols. The overall idea of the protocol is as follows. We make an agent (the cutter) divide the cake into n equally preferred pieces. The cake is then allocated to the agents with each agent getting a part of one of the pieces. The subroutine will be referred to as the *Core Protocol*. The Core Protocol itself relies on recursively applying the *SubCore Protocol*. When running the Core Protocol, if all the cake is already allocated, we are done. Otherwise, the Core Protocol is run repeatedly on the left-over cake. If some cake is still unallocated, we repeat it with another agent as cutter. After the Core Protocol has been run repeatedly, it is checked whether agents have discrepancy on a part of the cake. Discrepancy implies that agents have radically different valuations over two segments of the cake. This works to our advantage because we can run the *Discrepancy Protocol* in which we let the differing agents concentrate on the segments they prefer much more which simplifies our problem to envy-free cake cutting for a smaller number of agents. Otherwise, we exchange some parts of the cake that certain agents hold using the GoLeft Protocol in a systematic way to obtain further dominations between agents. After the GoLeft routine has been implemented, some agents dominate all other agents. At this point, we have decomposed the problem into a smaller problem with less number of agents. Hence the envy-free algorithm can be called recursively on the unallocated cake to divide the cake among the specified subset of agents.

A. *Core Protocol*

We first present the Core Protocol that calls the SubCore Protocol. The Core Protocol is helpful in allocating additional residue in an envy-free manner. The main challenge is that just by running the Core Protocol repeatedly on the residue, there is no guarantee that the cake will be allocated completely in bounded or even finite number of steps. Hence, we will introduce other protocols in addition to the Core protocol that help us allocate the whole cake in an envy-free manner.

Algorithm 1 Core Protocol

```
Input:  Specified cutter (say agent $i \in N$), agent set $N$ such that $i \in N$,
    and unallocated cake $R$.
Output: An envy-free allocation of cake $R' \subset R$ for agents in $N$ and
    updated unallocated cake $R \setminus R'$.
1: Ask agent $i$ to cut the cake $R$ into $n$ equally preferred pieces.
2: Run SubCore Protocol on the $n$ pieces with agents set $N \setminus \{i\}$ which
   gives an allocation to the agents in $N \setminus \{i\}$ such that one of the $n$ pieces
   is untrimmed and unallocated.
3: Give $i$ one of the unallocated untrimmed pieces from the previous step.
4: return envy-free partial allocation (in which each agent gets a connected
   piece) as well as the unallocated cake.
```

The Core Protocol asks a specified agent termed as the cutter to cut the unallocated cake into n equally preferred pieces. It then calls the SubCore Protocol that allocates to each of the agents one of the pieces (possibly partially) in an envy-free manner. No agent is given cake from any other piece. The name of the protocol is the same as the Core Protocol used by Aziz and Mackenzie [1] for the case of 4 agents which returns an envy-free partial allocation in which one agent cuts the cake into four equally preferred pieces and the cutter as well as at

Algorithm 2 SubCore Protocol

```
Input:  Cake cut into $n$ pieces to be allocated among agents in a set $N' \subset$
    $N$ with $n' = |N'|$.
Output: An envy-free partial allocation for agents in $N'$ in which each agent
    gets a connected piece.
1: Take permutation $1, 2, 3, \dots, n'$ as the order of agents in $N'$.
2: for $m = 1$ to $n'$ do
3:   we ask the $m$-th agent his most preferred piece among the $n$ pieces.
4:   if he has a most preferred piece that is not one of the tentatively
     allocated pieces then
5:     we can tentatively give the $m$-th agent that piece and go the next
       iteration of the 'for' loop.
6:   else the first $m$ agents are contesting for the same $m - 1$ tentatively
     allocated pieces. Then each agent in $[m]$ is asked to trim pieces among
     the $m - 1$ tentatively allocated pieces that are of higher value than his
     most preferred piece outside the $m - 1$ allocated pieces so that value of
     the former is same as the value of the latter.
7:     Set $W$ to be the set of agents who trimmed most (had the rightmost
       trim) in some piece. {$W$ is the set of agents who are guaranteed to have
       an envy-free allocation where each agent gets a piece that he trimmed
       most. In fact there exists an envy-free neat allocation for agents in $W$
       because one of the agents can take a most preferred full piece outside
       of the contested pieces. We will now try to increase the size of $W$.}
8:     while $|W| < m$ do
9:       Ignore previous trims of agents in $W$ from now on and forget
         the previous allocation.
10:      Run SubCore Protocol on all of the $n$ pieces with $W$ as the target
         set of agents and for each piece, the left side of the right-most trim by
         an agent in $[m] \setminus W$ is ignored.
11:      The result of SubCore is an allocation that gives a (partial) piece
         to each of the agents in $W$ and an agent $i \in [m] \setminus W$ can be given
         a most preferred piece that is full and unallocated: $W \longleftarrow W \cup \{i\}$.
         At this point the allocation due to the recursive call is tentative and not
         permanently made.
12:    end while
13:  end if
14:  Each agent can get the right hand side of the piece that he trimmed
     most. Such a piece is connected.
15: end for
16: return envy-free partial cake for agents in $N'$ (such that each agent gets
    a connected piece that is on the right hand side of the original piece he
    trimmed most) as well as the unallocated cake.
```

least one other agent gets one of these four pieces. The new Core Protocol can be considered as a useful generalization of Core Protocol for the four-agent algorithm of Aziz and Mackenzie [1]. Unlike the Core Protocol for the four-agent case, the new Core Protocol requires a more sophisticated recursive SubCore Protocol. The SubCore Protocol results in an allocation that satisfies some 'neat' properties.

Definition 1 (Neat Allocation). *Consider a cake divided into n pieces and $m < n$ agents. An allocation of the cake into m agents is* neat *if (i) each agent's allocation is a (not necessarily whole) part of one of the pieces; (ii) one piece is unallocated; (iii) no agent prefers an unallocated piece over his allocation; and (iv) no agent prefers another agent's allocation over his allocation.*

We can prove that the Core Protocol results in an envy-free partial allocation in which the cutter cuts the cake into n pieces, each agent gets a part of exactly one of the pieces, at least one non-cutter agent gets a complete piece and at least one piece is unallocated which no non-cutter agent envies over his allocation. Since two agents get full pieces, from the cutter's perspective, $2/n$ of the cake is allocated after an iteration of the Core Protocol.

Most of the technical work in the Core Protocol is done when it calls the SubCore Protocol. In the SubCore Protocol, the main idea is that we start from a single agent and gradually grow the number of agents in a specified order while making sure that a neat allocation exists for the growing set of agents $\{1, \dots, m\}$ in which at least one agents gets a full piece. We denote $\{1, \dots, m\}$ by $[m]$. All the allocations encountered during the course of the SubCore Protocol are considered tentative until the final step. If the next agent in the specified order most prefers a piece that is not currently (tentatively) allocated, we can easily handle the new agent as he can get the unallocated piece without causing envy for anyone. Otherwise, we have to do more work to ensure that the previously handled agents as well as the new agent can simultaneously get a neat allocation in which at least one agent gets a full piece. This includes calling the SubCore Protocol recursively for a smaller number of agents.

In the SubCore Protocol, if the m-th agent is also interested in one of the $m-1$ tentatively assigned pieces, then each agent in $[m]$ is asked to trim pieces among the $m-1$ tentatively allocated pieces that are of higher value than his most preferred piece outside the $m-1$ allocated pieces so that value of the former is the same as the value of the latter. The set of agents who have the rightmost trim in some piece is referred to as W. Note that W is the set of agents who are guaranteed to have an envy-free allocation where each agent gets a piece that he trimmed most. In fact there exists an envy-free neat allocation for agents in W because one of the agents can take a most preferred full piece outside of the contested pieces. The protocol then increases the size of W until $W = [m]$. W is expanded as follows. The previous trims of agents in W are ignored. Run the SubCore Protocol on all of the n pieces with W as the target set of agents and for each piece, the left side of the right-most trim by an agent in $[m] \setminus W$ is ignored. Note that by ignoring the trims of agents in W, we have more cake that could potentially be allocated than the allocation where the trims of agents in W are not ignored but each agent in W gets the right hand side of the piece where he had the rightmost trim. The result of the SubCore recursive call is an envy-free allocation that gives a (partial) piece to each of the agents in W and an agent $i \in [m] \setminus W$ can be given a most preferred piece that is full and unallocated. Thus W can always be expanded until $W = [m]$ in which case we exit the while loop. Due to the recursive nature of SubCore, it requires n^n queries. Given that SubCore is envy-free, Core is envy-free as well because the cutter gets a full piece and no one else is envious of the cutter. For $n = 2$, the Core Protocol coincides with the well-known Divide and Choose protocol.

B. Ground Work for the Main Protocol

We now prepare the ground work for the Main Protocol as well as the other protocols it calls by establishing important terms and concepts.

Definition 2 (Snapshots)**.** *When we run the Core Protocol we end up with an envy-free partial allocation of the cake and a residue. We call the partial allocation a* snapshot*.*

The algorithm will keep track of certain snapshots which we will label p_j with $j \in \{1, \dots, C'\}$. The pieces allocated in each snapshot p_j are labelled c_{jk} where $k \in \{1, \dots, n\}$ indicates which agent got the piece. Since the allocation in each snapshot is envy-free, each agent thinks he got at least as much value for his piece as any other allocated piece. Therefore each agent thinks he has some (possible zero or more) bonus value over another agent in the core snapshot. Our protocol will make use of these bonuses. In our protocol, we repeatedly call the Core Protocol over the resultant residue thereby making the residue smaller until the set of snapshots have enough structure that we will exploit later.

a) Parameters: We will use four parameters to represent the bounds we work with. These are $C \ll C' \ll B \ll B'$. These are constant once n is fixed, however they are dependent on n. Since the Main Protocol calls itself on a strict subset of the agents, we will use the notation B'_{n-1} to label the bound on the Main Protocol run on $n-1$ agents. The bounds correspond to the following concepts. C' is the number of snapshots generated by the main algorithm that we label and keep track of. All subsequent partial allocations generated by runs of the Core Protocol simply serve the purpose of making the residue smaller from the cutter's perspective. Once the main algorithm has extracted all the pieces it needs from the residue and if a discrepancy has not been successfully exploited, the GoLeft Protocol will be run. When we run GoLeft we look at a subset of C isomorphic snapshots from the C' total snapshots. The value B' corresponds to the total number of queries required to run the whole protocol. B is used to define what we call significant pieces or values. Running the Core Protocol guarantees that a piece of cake is made smaller from the perspective of the cutter. The value B is a bound on the number of times the algorithm allows us to run the Core Protocol to make the residue smaller.

For the sake of achieving our boundedness results, the following values of the parameters work. (i) $C = n^{n^n}$ (ii) $C' = n^{n^{n^n}}$ (iii) $B = n^{n^{n^{n^n}}}$ (iv) $B' = n^{n^{n^{n^{n^n}}}}$. We have not optimized the values of the bounds so we expect that our general algorithmic approach works for more optimal bounds.

In the protocol we often want to make a piece of cake (the residue) smaller. To do so we run the Core Protocol on it. To make descriptions more succinct we define a function which converts the number of times we run the Core Protocol on a piece of cake to how much smaller it is from the cutter's perspective. Note that when the Core Protocol is run, at least $2/n$ value of the cake is allocated to the agents because the cutter cuts the cake into n equally preferred pieces and at least two pieces are fully allocated. By running the Core Protocol B times, the cutter thinks only $f(B) = (\frac{n-2}{n})^B$ value of the cake is unallocated.

Definition 3 (Bound function f)**.** $f(B) = (\frac{n-2}{n})^B$.

For any piece of cake c in a given snapshot, each agent has a bonus value which corresponds to how much more cake he thinks he got in that snapshot than he would have got had he been allocated c instead.

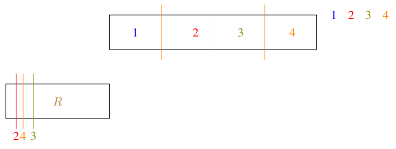
Definition 4 (Bonus value)**.** *An agent i's* bonus value *on piece of cake c_{jk} in snapshot p_j is the value $V_i(c_{ji}) - V_i(c_{jk})$, where c_{ji} is the piece that was allocated to agent i in p_j by the Core Protocol.*

In a core snapshot, a bonus value is *significant* if we can make the residue smaller than that value from each agent's perspective in a bounded number of steps. Note that the significance of a piece of cake does not depend on the absolute value that an agent ascribes to it but it is relative to the value of the unallocated cake (residue).

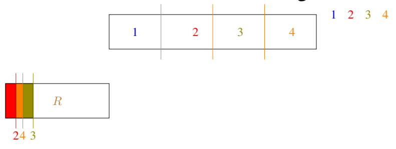
Definition 5 (Significant value)**.** *An agent i thinks a value is* significant *if the value is more than or equal to $V_i(R)f(B)$ where R is the unallocated residue. A piece is significant for an agent if it has significant value for him.*

Note that if an agent i finds a piece of cake significant, he will still find it significant if the residue becomes smaller than before. We also observe the following about the core protocol and the cutter getting a significant advantage over at least one other agent.

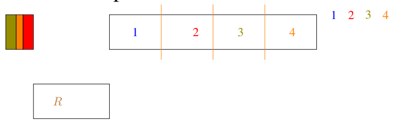
Remark 1. *When the Core Protocol is run once, the cutter has a significant bonus over at least one agent (the agent who gets the smallest value piece from the cutter's perspective). This bonus results in a domination of the cutter over the agent in $k = (\log n)(\frac{n-2}{n}) + 1$ iterations of the Core Protocol on the residue with same cutter.*



(a) Agents 2, 3, 4 first make trims on the residue so that left side of the trim is of same value as their bonus over agent 1 in the core snapshot.



(b) The trims of the agents on the residue result in a partition of the residue into small pieces.



(c) The pieces resulting from the trims are extracted from the residue and associated with the corresponding piece.

Fig. 2: Process of extraction for the piece allocated to agent 1 in the core snapshot.

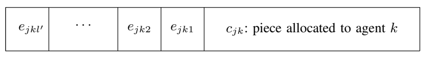


Fig. 3: For any allocated piece in some core snapshot, we consider extracted pieces to be attached to the piece. In the figure, piece c_{jk} is allocated to agent k in snapshot p_j. The pieces $e_{jk1}, e_{jk2}, \ldots, e_{jkl'}$ are extracted from the residue and are in consideration for attachment to the piece c_{jk}.

If we run the Core Protocol repeatedly with the same agent as cutter and we are lucky that the cutter has a significant advantage over each agent in some snapshot, then in $(\log n)(\frac{n-2}{n}) + 1$ extra iterations of the Core Protocol with the same cutter, we can make the cutter dominate each agent. This simplifies our problem, because we now only need to allocate the remaining cake among the dominated agents in an envy-free manner. In general, we may not be so lucky that the cutter dominates all other agent due to which we have to do more work to ensure that a set of agents dominates the other agents. For this, the process of extraction is crucial:

b) Extraction: For each piece allocated in each of the core snapshots and for each agent, the Main Protocol tries to associate a piece of the cake extracted from the residue. For each of the C' snapshots, there are $C'n$ pieces allocated to the agents. For each of the pieces, all the agents *extract* corresponding pieces from the residue that is the unallocated cake. Take for example, a piece c allocated to some agent (say agent 1) in a snapshot. In the snapshot in which c is allocated to 1, each agent i gets a piece that is of value (according to i) at least as much as c. In the residue, each agent i is asked to put a trim so that cake from the left extreme of the residue to the trim is of value equal to i's value of his piece minus his value for c. The trims of the agents on the residue give rise to pieces of cake corresponding to the agents' bonus value over piece c. These pieces are *extracted* (cut away from the residue) and *associated with* c (kept in consideration with piece c). When agents make trim marks on the residue in accordance with their bonus (over a piece in a snapshot), each pair of successive trim marks gives rise to a separate piece that can be extracted. For piece c_{jk} in snapshot p_j, we will denote by $e_{jk1}, e_{jk2}, \ldots, e_{jkl'}$ the set of pieces that are extracted in the same order with e_{jk1} as the leftmost in the residue and which is extracted first (see Figure 3). Note that $l' \leq n-1$ because for each allocated piece in a snapshot, at most $n-1$ other agents can put trim marks on the residue so as to obtain $n-1$ extracted pieces. We say that a piece e_{jkl} is extracted by agent i if during extraction, the right hand extreme of the piece coincided with the trim of agent i on the residue. Each extracted piece has a clear corresponding allocated pieces in a core snapshot with which it is associated. In our protocol, pieces are extracted from the residue only if each agent finds the pieces *not* significant. The reason is that we want to keep sufficient residue to do further extractions on the residue as well as maintain structure.

Later on, the extracted pieces may be attached to c so that piece c is now attractive to other agents because of the additional extracted pieces combined with c. We clarify that when pieces are extracted and associated with a given piece in a core snapshot, such extracted pieces have not yet been allocated to any particular agent. Extracted pieces can only be allocated after they are officially attached to their associated piece. For the protocol we need to restrict our focus to a subset of the snapshots where the same set of agents extracted pieces in the same order:

Definition 6 (Isomorphic snapshots/snapshot pieces/ extracted pieces)**.** *We call a set of snapshots* isomorphic *to each other if for each piece c_i in the snapshot allocated to agent i, the set of agents who extracted cake from the residue and associated to c_i are the same and did so in the same order. We extend this notion to allocated pieces in snapshots. Two allocated pieces of cake belonging to two isomorphic snapshots are isomorphic if they were allocated to the same agent. We also extend the notion of isomorphism to extracted pieces. We say that for two isomorphic snapshots p_j and $p_{j'}$, two extracted pieces e_{jkl} and $e_{j'kl}$ are isomorphic if they are associated respectively to isomorphic pieces c_{jk} and $c_{j'k}$ and that e_{jkl} and $e_{j'kl}$ were extracted by the same agent.*

Our protocol will keep track of a set of isomorphic snapshots, progressively discarding (not changing) some so that we can make manipulations on the ones we keep in an envy-free way. The pieces of cake we are working with are labelled c_{jk} for allocated pieces or e_{jkl} for extracted pieces. We use the notation c_k, S and e_{kl}, S to denote the set of pieces of cake c_{jk} or e_{jkl} in snapshots in S. Abusing the notation, we will simply say c_k or e_{kl} to mean the set of pieces which are in the snapshots we are working with. Note that we will generally use index j for the snapshot number, k for the piece number in a given snapshot, and l for the l-th extracted piece.

C. The Main Protocol

The Main Protocol is the engine which runs our overall envy-free protocol. It is responsible for allocating the whole cake among all the agents in an envy-free manner.

The Main Protocol works recursively. If the number of agents is four or less, a previously known bounded envy-free algorithm can directly be called. Otherwise, the Main Protocol divides part of the cake in an envy-free manner and identifies a set of agents $N \setminus A$ that all dominate agents in A wrt the cake that is unallocated. The Main Protocol then recursively calls itself to divide the remaining cake among agents in A.

If the number of agents is more than four, the Main Protocol (Algorithm 3) calls the Core Protocol sequentially on the updated residue so that the residue becomes smaller than before after each call of the Core Protocol. The repeated calls of the Core Protocol help generate a number of snapshots each containing n pieces of cake. In each snapshot, each agent has been allocated a piece of cake. Envy-freeness is maintained throughout for the allocated cake, and in the Core Protocol each agent thinks he got the highest value piece.

Algorithm 3 Main Protocol

Input: A cake R and a set of agents N with $n = |N|$.
Output: An envy-free allocation that completely allocates R among agents in N.

```
Base Case
1:  if $|N| \le 4$ then
2:    allocate the residue among the agents in $N$ in an envy-free allocation by using a known constant-time envy-free protocol for $|N| \le 4$.
3:  else
Generate Core Snapshots
4:    for all $j : 1 \ldots C'$ do
5:      For some $i \in N$ who has acted as cutter in the Core Protocol least number of times, run Core protocol($i$, $N$, $R$); Update $R$ to the cake that is unallocated. The Core Protocol allocation gives us snapshot $p_j$ and $n$ pieces of cake $c_{jk}$ with $k : 1, \ldots, n$.
6:    end for {We have now generated $C'$ core snapshots. After generating the first $n$ snapshots, we already have an envy-free allocation in which each agent gets $1/n$ value of the original cake.}
7:    while for agent $i \in N$ and some piece $c_{jk}$ $V_i(R)f(B)^2 < V_i(c_{jk}) < V_i(R)f(B)$ do {We ensure that the values are either significant or smaller than significant by a large factor dependent on our bound $B$.}
8:      Run Core Protocol($i$, $N$, $R$).
9:    end while
10:   if some set of agents $N \setminus A$ dominates all agents in set $A$ then Call Main Protocol($R$, $A$) to divide the unallocated cake $R$.
11:     return allocation of the cake to the agents.
12:   end if
Extraction
13:   Define Boolean $a$ and set it to 0 {This Boolean is used to reset the whole trimming process when running the Discrepancy Protocol causes a bonus value to become significant.}
14:   while $a = 0$ do
        {We now attempt to extract pieces corresponding to the non-significant bonus values from the residue, resetting the whole process when a discrepancy in agents' valuations has forced us to shrink the residue}
15:     Set $a$ to 1.
16:     for all pieces of cake $c_{jk}$ do
17:       for all agents $i$ do
18:         if $i$'s bonus value is not significant on piece $c_{jk}$ then
19:           Ask $i$ to place a trim on the residue such that the piece stretching from the left of the residue to the trim is equal to the agent's bonus value for that piece, which we will label $b^i_{c_{jk}}$.
20:         end if
21:       end for
22:       Label from left to right the pieces delimited by those trims with labels $e_{jkl}$ with $l : 1, \ldots, m$ where $m$ is the number of agents who placed a trim on the residue (did not have significant bonus) and $j$ and $k$ are used as previously {if two or more agents' trims coincide we break ties lexicographically and ascribe empty cake to the latter agents' extractions.}
        {For a given piece of cake $c_{jk}$ the $m$ agents with non-significant bonus value have now delimited a piece of cake corresponding to that value from the residue, and those pieces have been labelled.}
23:       for all $l : 1, \ldots, m$ do
24:         if no agent thinks the piece $e_{jkl}$ is significant then { Everyone agrees piece is insignificant.}
25:           Extract piece $e_{jkl}$ from the residue, associate it to piece $c_{jk}$ and update $R$ to $R - e_{jkl}$ {See Figure 2}.
26:         end if
```

Remark 2. *When we call the Core Protocol n times each time with a different cutter, each agent gets at least $1/n$ of the value of the cake allocated in that call. Moreover when an agent himself is the cutter, he gets at least $1/n$ of the value of the remaining cake. Hence after the Main Protocol has made the first n calls of the Core Protocol, we have an envy-free allocation in which agents get at least $1/n$ value of the original*

```
27:            {continued}
Discrepancy
28:            if some agents think $e_{jkl}$ is significant but others do not (agents think the piece is discrepant) then
                 {There is a large discrepancy between agents' valuation, this causes a problem because some agents might think we are taking too much away from the residue. However discrepancy can be exploited or eliminated.}
29:              Run Discrepancy($e_{jkl}$,$b^u_{c_{jk}}$,$\{e_{jkl}\}$,$R$) with the discrepant piece $e_{jkl}$, the bonus value $b^u_{c_{jk}}$ of the agent $u$ who made the rightmost trim delimiting the piece, the set of previously extracted pieces of cake $\{e_{jkl}\}$ and the residue $R$ as input. This will return the Boolean value DISCREPANCY and when that value is 1 the protocol also returns two sets $D \subset N$ and $D' \subset N$ such that $D$ and $D'$ partition $N$.
30:              if DISCREPANCY$=$ 1 then {The discrepancy can be exploited.}
31:                Main Protocol($e_{jkl}$, $D$).
32:                Main Protocol($R$, $D'$).
33:                return allocation of the cake to the agents.
34:              else{The discrepancy cannot be exploited but $b^u_{c_{jk}}$ is now a significant bonus value.}
35:                Add $e_{jkl}$ to $R$ and Set $a$ to 0.
36:              end if
37:            end if
38:          end for
39:        end for
40:      end while
GoLeft
41:      Run GoLeft($N, \{p_j\}$, $\{c_{jk}\}$, $\{e_{jkl}\}$, $R$) Protocol with the set of labelled snapshots $\{p_j\}$, the set of allocated pieces of cake $\{c_{jk}\}$, the set of extracted pieces $\{e_{jkl}\}$ and the residue $R$ as input. The output of GoLeft is a set $A \subset N$.
42:      for all agents $i$ do
43:        Run Core Protocol($i$,$N$,$R$) $2B$ times each time on the updated smaller residue $R$.
44:      end for
           {The previous for loop converts significant bonus values into dominance wrt the residue $R$.}
Recursion of the Main Protocol
45:      Call Main Protocol($R$, $A \subset N$), that is the Main Protocol on a subset of agents $A \subset N$ output by the GoLeft Protocol and with $R$ as the input cake.
46:    end if
47:    return allocation of the cake to the agents.
```

cake.

After the first n calls. the Main Protocol does not stop making calls to the Core Protocol if there is still some unallocated cake. After calling the Core Protocol C' times, C' core snapshots are obtained. The Core Protocol may be further called (in the while loop in step 7) to make the residue even smaller. This ensures that each agent considers each piece in the first C' snapshots significant or smaller than significant by a large factor dependent on our bound B.

For each piece of cake c in the C' snapshots, agents can ascribe what we refer to as a *bonus value*, which corresponds to how much more value they got in that snapshot than their value for piece c. In the Main Protocol, for each piece of cake c in the snapshots, we ask all agents with a non-significant bonus value to make a cut on the residue equal to their bonus value. The pieces obtained from these cuts are then taken from the residue and associated (but not yet attached) to piece of cake c. We refer to this process as extraction. A piece is extracted only if all agents find it insignificant. The extracted pieces will potentially be attached to their associated piece of cake c in the GoLeft Protocol. Most of them however will be sent back to the residue or shared amongst a subset of agents in an envy-free way. The process of attaching the extracted pieces to their associated piece in the GoLeft Protocol is to make that piece desirable to the agent whose bonus value was used to extract the piece from the residue. The Main Protocol also calls the Discrepancy Protocol in case there is some piece in consideration for extraction that some agents consider significant and others do not. The goal of the Discrepancy Protocol is to exploit any such discrepancy and to ensure that when the GoLeft Protocol is called by the Main Protocol, then there is no discrepancy in how the extracted pieces are viewed i.e., no actually extracted piece is considered significant by some agent.

D. *Discrepancy Protocol*

When pieces are being extracted from the residue during the Main Protocol, it may be the case that one of the pieces e_{jkl} in consideration for extraction is significant for some agent. In that case, the piece is not extracted and the Discrepancy Protocol (Algorithm 4) is called in line 29 that either exploits or 'eliminates' this discrepancy. The discrepant piece e_{jkl} is kept aside from the residue. On the other hand, all previously extracted pieces are added back to the residue. Since the difference between a piece that is just above significant or just below significant can be arbitrarily small, the Core Protocol is used to create a *gap* so that the discrepant piece either has value at least $V_i(R)n$ or value at most $V_i(R)/n$. This gap is highly useful because the goal of the Discrepancy Protocol is to either ensure that (1) everyone thinks that the discrepant piece is significant or (2) the problem of finding an envy-free allocation can be broken into two sub-problems where some agents D are allocated the discrepant piece and the rest D' are allocated the residue. In this, we use the fact we mentioned earlier that envy-freeness implies proportionality.

In case of (1), the process of extraction in the Main Protocol is reset and the discrepant piece as well as all the pieces that had been extracted are sent back to the residue. This may appear to be a waste of work but when we do this, we have ensured that at least one agent has significant advantage over another's piece for a given snapshot. This 'setback' can only happen $C'n^2$ times before each agent dominates each other agent in which case the remaining cake can be allocated arbitrarily without causing envy.

We note that Discrepancy makes the residue smaller as it calls the Core Protocol. When the residue becomes smaller, it may be that a piece of cake that was not significant for an agent becomes significant because significance is defined wrt the residue. Hence any agent i who thinks that $\frac{V_i(R)}{n} \le V_i(e_{jkl}) \le V_i(R)n$ will eventually think that $V_i(e_{jkl}) \ge V_i(R)n$ when the Core Protocol has been run to reduce the residue. Note that the Discrepancy Protocol is envy-free as long as the Core Protocol is envy-free. The Discrepancy Protocol is bounded by $Bn \times n^n$.

Algorithm 4 Discrepancy Protocol

```
Input:  Residue $R$, discrepant piece $e_{jkl}$, bonus value $b^u_{c_{jk}}$ on $c_{jk}$ of
        the agent $u$ who wanted to extract the discrepant piece (but could not
        extract), set of extracted pieces of cake $\{e_{jkl}$ : extracted pieces$\}$ , and
        agent set $N$.
Output: Possibly modified residue $R$, a Boolean value called DISCREPANCY,
        a set of agents $D$, and a set of agents $D'$.
 1: Take out the discrepant piece $e_{jkl}$ from $R$; Reinsert all the extracted
    pieces back into the residue, relabel the aggregate piece $R$.
Run Core
 2: while for some agent $i$, it is the case that $\frac{V_i(R)}{n} \leq V_i(e_{jkl}) \leq V_i(R)n$
    do
 3:     Run Core Protocol($i$, $N$, $R$) $B$ times iteratively on the updated residue
        $R$.
 4: end while
Exploit Discrepancy
 5: if $e_{jkl}$ still has some agents consider it significant and others not then
    {In this case we can exploit the discrepancy and 'separate' the agents.}
 6:     Set $D$ to $\{i \in N : V_i(e_{jkl}) \geq V_i(R)n\}$.
 7:     Set $D'$ to $\{i \in N : V_i(e_{jkl}) \leq \frac{V_i(R)}{n}\}$. {Note that $N = D \cup D'$.}
 8:     Set DISCREPANCY to 1.
 9:     return $R$, DISCREPANCY, $D$ and $D'$.
10: else
Cannot Exploit Discrepancy — but all agents now on the same page
        { This means that we cannot exploit a discrepancy but now all agents
        think that the bonus value of whoever made the trim delimiting the
        discrepant piece is significant, including agent $u$ who made the trim.}
11:     Set DISCREPANCY to 0.
            {$b^u_{c_jk}$ is now considered significant by agent $u$.}
12:     return $R$ and DISCREPANCY.
13: end if
```

E. Groundwork for the GoLeft Protocol

The GoLeft Protocol is the heart of our overall protocol and is crucial to allocate the cake that is still not allocated. All the other steps in the Main Protocol can be viewed as preparing the ground for the GoLeft Protocol to work. By calling the Core Protocol sufficient number of times in the Main Protocol, enough C' core snapshots are obtained that are helpful to identify C isomorphic snapshots in the GoLeft Protocol (these isomorphic snapshots constitute the working set of snapshots over which the GoLeft Protocol does further operations). Moreover, by calling Discrepancy before GoLeft, it is ensured that all agents are on the same page: all agents consider all the extracted pieces as insignificant.

The GoLeft Protocol (Algorithm 5) is called by the Main Protocol. The goal of the GoLeft Protocol is to identify a set of nodes $N \setminus A$ that dominate agents in A. This means that the remaining residue can be allocated among agents in A in an envy-free manner without worrying about agents in $N \setminus A$ envying them. The goal of the GoLeft protocol is achieved by attaching extracted pieces to the pieces in the working set of snapshots in a methodical manner while maintaining envy-freeness of the allocated cake. In order to maintain envy-freeness, the working set of snapshots is modified in various ways. The main thing is that when all extracted pieces for the working set of snapshots have been attached, we can identify a set of agents that all dominate the other agents. For example, if we end up with core snapshots in which for one isomorphic allocated piece, there are a total of less than $n-1$ extracted pieces that have all been attached and are held by a certain agent i, then agents who did not manage to extract pieces corresponding to the main piece have a significant advantage over i as well as all other agents who extracted pieces before i for that main piece. This significant advantage translates into dominance.

The GoLeft Protocol makes operations on a working set of isomorphic snapshots. The main operations of the GoLeft Protocol are to attach extracted pieces to the core snapshot allocation and to implement exchanges in which there is sequence of agents $a_o, a_1, \ldots, a_{k-1}$ where each agent a_i in the sequence gets the pieces of agent $a_{i+1 \mod k}$. When operations such as attachments and exchanges happen, the isomorphic snapshots change but remain isomorphic nonetheless.

The GoLeft Protocol involves two key mathematical structures: a *working set S of isomorphic snapshots* and the *permutation graph* that is defined wrt the working set of isomorphic snapshots. During the GoLeft Protocol, each structure gets updated based on the information provided by the other structure.

a) Working Set of Isomorphic Snapshots: During the course of the GoLeft Protocol, the isomorphic snapshots in S get changed in the following way: (1) some subset of S is removed from S (2) agents exchange their pieces and (3) extracted pieces are attached to pieces in the snapshots in S.

When we update S, we maintain isomorphism and other invariant properties as follows. If an agent holds a piece c_{jk}, he holds the whole set c_k of isomorphic pieces in S. If he holds an extracted piece e_{jkl}, he holds the whole set e_{kl} of isomorphic extracted pieces associated with S as well. When we implement an exchange, we are essentially making $|S|$ exchanges – one in each of the snapshots. These are exchanges that not only involve the pieces in the snapshots but also involve those extracted pieces that have been *attached* to the pieces. When we attach an extraction to a piece in a snapshot in S, we simultaneously attach isomorphic extractions to the corresponding isomorphic allocated piece in each of the snapshots.

We also ensure that the extracted pieces are attached in the appropriate order so that each associated piece in $e_{k(l+1)}$ gets attached after the previous associated pieces have been attached. For example, if we were focusing on the snapshot in Figure 2, the extracted associated piece due to agent 3 will get attached after the extracted associated piece due to agent 4. Note that if we attach an extracted piece e_{jkl} to an allocated piece c_{jk} (along with its previously attached extracted pieces) in a snapshot in the working set S, we perform a similar attachment for all such isomorphic extracted pieces in the set e_{kl} to their corresponding pieces in set c_k. Also, if an agent i currently holds an extracted piece, he also holds all earlier extracted pieces as well. We remark about a consistency condition that we enforce.

Remark 3. *We enforce a consistency condition whereby an agent cannot hold extracted pieces beyond the extraction he himself made. Therefore, when we attach an extracted piece* *to agent j's piece in a snapshot to attract agent i to it, agent j does not actually hold the latest attachment because it is beyond j's extraction. However in an exchange, if i were to get j's piece along with its attachments, then i will also get the latest attachments that were originally extracted by i himself.*

In the GoLeft Protocol, we operate on an increasingly small set of isomorphic snapshots. The goal is to reach a set of isomorphic snapshots that can be used to find a set of agents who all dominate other agents. As the GoLeft Protocol proceeds, we discard snapshots to allow us to focus on others where extracted pieces from E have been 'attached' in an envy-free way to the allocated pieces in C to which they were associated. The set of isomorphic snapshots S becomes smaller when a set of isomorphic extractions $e_{k(l+1)}$ are attached to set of isomorphic pieces c_k in the snapshots S with each particular extracted piece in $e_{k(l+1)}$ getting attached to its corresponding piece in e_k. By *discarding* a set a core snapshots, we mean that these snapshots and their allocations are not further worked upon and their associated unattached extracted pieces are sent back to the residue. Intuitively, the purpose of discarding snapshots will be to preserve the remaining advantages of agents over other agents.

The GoLeft procedure gradually attaches the extracted pieces to the allocated pieces in the working set of snapshots. The reason we call the protocol 'go left' is because we can visualize that for each piece in the set of snapshots we work with, we want to add the next extracted pieces to the isomorphic allocated pieces that are kept to the left of the isomorphic allocated pieces. By attaching the next extracted pieces, we are 'going left'.

b) Permutation Graph: The permutation graph keeps track of which agent is willing to move to which piece in the snapshots. The high level idea is that nodes of the permutation graph correspond to the agents and an agent i points to another agent j if he will be as happy taking j's allocated pieces along with the attachments on those pieces.

Definition 7 (Permutation graph)**.** *The permutation graph is a directed graph where the set of nodes correspond to the set of agents. Hence when we refer to the graph, we will use agents and nodes interchangeably. The arcs of the permutation graph depend on the current state of the working set of isomorphic snapshots S. In particular, they depend on which extracted pieces have been attached to the originally allocated pieces in the isomorphic snapshots in S. Agent i points to agent j if j holds isomorphic pieces in S that have had all attachments up till i's extracted pieces. We build the initial permutation graph with each node pointing only to itself. Throughout the protocol, we ensure that each node in the permutation graph has in-degree at least one.*

The permutation graph itself gets updated when isomorphic extractions are attached to isomorphic pieces in S. The process of extracted pieces being attached to an allocated piece results in the aggregated piece becoming attractive to a new agent who then wants to point to the agent holding that piece.

The permutation graph also suggests a natural way to exchange pieces. It has a similar idea as the trading graph used in top trading cycles algorithm for housing markets. If there is a cycle in the graph, we have the possibility of exchanging the allocations of agents in the cycle by giving an agent the piece of the agent he points to. The permutation graph is more intricate because updates on the permutation graph reflect simultaneous updates on the working set of isomorphic snapshots S. Also by Remark 3, an agent in an exchange does not offer what he holds but can offer more because of the additional attachments beyond his own extraction.

F. The GoLeft Protocol

When the GoLeft Protocol starts, it first identifies a working set S of C core snapshots from out of the C' core snapshot that we focus on. The protocol then constructs a permutation graph corresponding to the working set of isomorphic snapshots.

In the permutation graph, each node i corresponds to an agent i who holds a set of isomorphic pieces along with their attached extracted pieces in the working set of isomorphic snapshots S. We divide the nodes of the permutation graph into sets T and T'. Set T is the set of nodes/agents such that the isomorphic pieces held by them in S have not had $n-1$ attachments). T' is the set of nodes/agents such that the isomorphic pieces held by them in S have had $n-1$ attachments.

The protocol identifies a cycle in the permutation graph that includes at least one node i from T. Such a cycle always exists. In each of the working set S of isomorphic snapshots, we implement an exchange of pieces held by agents in the cycle: each agent in the cycle is given the piece corresponding to the node that the agent points to in the cycle. After implementing the exchange, the permutation graph is updated to reflect the exchange. If in the exchange, an agent now gets isomorphic pieces that he thought were inferior, he always gets the additional extracted pieces associated with the inferior piece up till the agent's extractions. Hence, each agent's total value in each isomorphic snapshot in S stays the same. For any agent i, as long as no agent gets extracted pieces beyond i's extraction, i will not be envious. In the GoLeft protocol, it can be the case that some agent j gets extracted pieces beyond i's extracted pieces but before any such attachments in the last part of the GoLeft protocol, we ensure that no envy arises.

After implementing the cycle, we focus on a node $i \in T$ that was in the cycle. For agent/node i we know that for all snapshots in the working set S, agent i has been allocated the original isomorphic pieces c_k as well as all associated pieces up till i's extracted piece. If the piece of cake agent i is currently allocated in the snapshots S has no more extracted pieces left to attach to it, but it has not had $n-1$ attachments, this means that all agents who have not had their corresponding piece extracted/attached have a significant advantage over agents who have had an extracted piece attached. In this case, the GoLeft Protocol returns the set of dominated agents to the Main Protocol and we are left with a smaller envy-free allocation problem because it involves less number of agents.

Algorithm 5 GoLeft Protocol

```
Input: A set $N$, a set $C'$ of snapshots $p_j$, a set of corresponding pieces
  $\{c_{jk}\}$, a set of extracted pieces $\{e_{jkl}\}$, and a residue $R$.
Output: A set of agents $A \subset N$ such that all agents in $N \setminus A$ dominate all
  agents in $A$.

Isomorphic Snapshots and build Permutation Graph
 1: Select from the $C'$ snapshots a set of size $C$ of isomorphic snapshots
 2: Relabel the $C$ snapshots $p_j$ with $j$ now ranging from 1 to $C$.
 3: Declare set $S$ in which we add all $C$ snapshots $\{p_j : j \in \{1, \dots, C\}\}$.
 4: Build the permutation graph with nodes corresponding to the agents.
    Throughout the algorithm, we maintain two sets: (1) $T$ (set of
    nodes/agents such that the isomorphic pieces held by them in $S$ have
    not had $n-1$ attachments and (2) $T'$ (set of nodes/agents such that the
    isomorphic pieces held by them in $S$ have had $n-1$ attachments). $T$
    and $T'$ partition the nodes. Initially, (1) all nodes are placed in $T$ so that
    $T'$ is empty; (2) each node points to itself in the permutation graph (3)
    each agent owns his originally allocated set of pieces allocated in the
    core snapshots in the working set $S$. Thus each node has the following
    associated information: agent and his allocation in snapshots in $S$.
 5: while there is a node in $T$ do
Cycle and Exchange
 6:   Find a cycle in the permutation graph which involves a node from $T$.
      {Since nodes in $T$ all have in-degree exactly 1 and each node points to
      each node in $T'$, such a cycle exists}
 7:   For the agents in the cycle, exchange the allocation of the agents in
      the snapshots in $S$ as follows: if $i$ points to $j$, give $i$ the pieces of cake
      that $j$ was currently allocated in snapshots $S$.
 8:   If $i$ points to $j$ in the selected cycle, do as follows. Transfer all in-
      edges of $j$ to $i$. Replace $j$ in set $T$ or $T'$ by $i$. { This update reflects
      the permutation of the allocation. At this point, each node that was in
      the cycle has a self-loop. }.
 9:   Take a node/agent $i$ in the cycle that is from $T$.
Separation - the only way the while loop exits
10:   if there is a node in the located cycle that is from $T$ (has had less
      than $n-1$ attachments) but has no extracted pieces to be attached then
11:     Focus on the set of isomorphic pieces $C$ associated with the node.
        All agents who have been given an element of $C$ are placed in $A$.
12:     return $A$.
13:   else we now know that there is an agent $i$ that was in the cycle from
      $T$ that holds pieces in $S$ that still have extracted pieces to be attached.
      We focus on this agent $i$ in the attachment phase.
14:   end if
Attachment — (to be done in a subset of the snapshots)
15:   Index the agents in the order which they extracted the pieces
      associated to $c_k$ so that 1 originally got $c_k$, 2 made the next extraction
      and so on. In doing so, we index agent $i$ as $l$.
        {We now attach in a subset of the snapshots the set of isomorphic
      extracted pieces in $e_{k(l+1)}$ to the set of pieces $c_k$, thus making pieces
      in set $c_k$ desirable to the agent who extracted pieces in $e_{k(l+1)}$. The
      set of pieces $c_k$ in S and its current attachments are currently owned
      by agent l. At this point for each isomorphic piece $c_k$, associated pieces
      up till l's trim have already been attached.}
16:   Declare $S' \leftarrow \emptyset$.
Attachment — making it agreeable for agents from $l+1$ to $n$
17:   for agent $i$ ranging from $l+1$ to $n$ do
18:     Ask $i$ to choose $\frac{|S|}{n-l+1}$ snapshots for which $i$ values the difference
        between his bonus value for $c_k$ and the extracted pieces currently
        attached to $c_k$ the most.
19:   end for
20:   Add all chosen snapshots to $S'$.
21:   All extracted pieces that have not been attached from snapshots in $S'$
      are put back into the residue $R$ and will never be attached. The new
      aggregate piece is labelled $R$.
22:   $S \leftarrow S \setminus S'$; $S'' \leftarrow \emptyset$.
Attachment — making it agreeable for agents from 1 to $l$
23:   for all agents $i$ ranging from 1 to $l$ do
24:     Ask $i$ to choose the $\frac{|S|n}{ln+1}$ snapshots for which he values the piece
        $e_{k(l+1)}$ the most.
25:     Add those to $S''$.
26:     {Continued}
27:   end for
28:   $S \leftarrow S \setminus S''$.
29:   All extracted pieces that have not been attached from snapshots in $S''$
      are put back into the residue $R$. The new aggregate piece is labelled $R$.
      {Again extracted pieces that will never be attached are sent back to the
      residue}
30:   All pieces $e_{k(l+1)}$ in snapshots in $S''$ are aggregated into a piece $a$.
31:   Run Main Protocol($a$, $\{1, \dots, l\}$).
32:   $S'' \leftarrow \emptyset$.
Attachment — now happening
33:   We now attach the piece of cake $e_{k(l+1)}$ to $c_k$ in the snapshots still
      in $S$, meaning that if agent $l+1$ were to move to piece $c_k$, he would
      also get $e_{k(l+1)}$. In other words $c_k$ is now desirable to $l+1$ because
      of the attachments. However in snapshots where we keep agent $l$ as the
      agent allocated piece $c_k$, the piece $e_{k(l+1)}$ is sent back to the residue,
      as it could not be given to $l$ in an envy-free way { This update reflects
      that we made a set of allocated pieces desirable to a new agent}.
34:   Remove the self-loop of $i$. Replace it with an edge going from the
      agent who extracted the piece that the protocol just attached to $c_{jk}$.
35:   If the pieces held by $i$ have had $n-1$ attachments, delete $i$ from $T$
      and place it in $T'$ and make every node point to $i$.
36: end while
```

In case node i does not lead to an exit from the GoLeft Protocol, we know that there are associated pieces that can still be attached to the isomorphic pieces held by i in the working set of core snapshots S. We focus on the next set of associated pieces $e_{k(l+1)}$ that we are interested to attach to the pieces c_k that have already had associated pieces $e_{k2}, e_{k_3}, \dots, e_{kl}$ attached in their corresponding main pieces c_k. Additionally attaching pieces $e_{k(l+1)}$ to pieces c_k is useful in making the agent who extracted them interested in the pieces c_k because of the additional $e_{k(l+1)}$ as well as the previous attachments. However, naively attaching the pieces can be problematic and spoil the envy-freeness of the allocation. We deal with the issue as follows.

The agents who did not extract pieces associated with the c_k pieces as well as agents who extracted pieces that have not been attached are asked to 'reserve' a big enough subset $S' \subset S$ of snapshots in which they value the difference between their bonus value for c_k and the extracted pieces currently attached to c_k the most. These snapshots S' are removed from S and their remaining unattached associated pieces are sent back to the residue. By maintaining the advantages in the snapshots S', such agents will not be envious even if some agent in $\{1, \dots, l\}$ additionally gets the remaining pieces $e_{k(l+1)}$.

The agents indexed from 1 to l who have all already had their extracted pieces attached to c_k are asked to choose a high enough fraction of the snapshots in which they value the $e_{k(l+1)}$ pieces. We call these snapshots S''. The $e_{k(l+1)}$ pieces from S'' are bunched together and the Main Protocol is called to divide this cake in an envy-free way among the agents indexed from 1 to l where l is strictly less than n. Since envy-freeness implies proportionality, they derive enough value that they will not be envious if agent indexed $l+1$ or higher gets all other pieces in set $e_{k(l+1)}$. The corresponding set of snapshots S'' are then discarded. Hence each time we attach isomorphic extracted pieces $e_{k(l+1)}$ to isomorphic pieces c_k, we discard snapshots $S' \cup S''$ from the working set S and maintain an overall envy-free allocation.

When the protocol attaches extracted pieces $e_{k(l+1)}$ to allocated pieces in c_k currently held by agent l, it deletes the incoming edge of node/agent l and replaces it by an edge coming from agent $l+1$ who extracted pieces in $e_{k(l+1)}$. Intuitively, $l+1$ is now willing to be allocated pieces in c_k and its attached pieces instead of his current pieces in S. We delete previous edges to ensure that until termination, nodes in T have in-degree strictly 1 which guarantees that no matter the cycle involving a node in T found by the protocol, we will make progress towards termination. By attaching enough extracted pieces in the appropriate order the GoLeft Protocol finally arrives at a point where there is some isomorphic set of pieces c_k in the set S for which all possible associated pieces have been attached but there is some set of agents $N \setminus A$ who do not have associated pieces. The reason agents in $N \setminus A$ could not extract such pieces is because they have a significant advantage over pieces in c_k in each initial isomorphic snapshot. By gradually attaching (unanimously insignificant) associated pieces to pieces in c_k and ensuring that all agents who did extract corresponding pieces do get some isomorphic piece in c_k (along with the associated insignificant attachments), we make sure that agents in $N \setminus A$ now dominate agents in A. At this point, we can return from the GoLeft Protocol. The GoLeft Protocol is bounded by $B'_{n-1} + 2Cn^3 + 2n^2 + C'$.

IV. Conclusion

By analyzing the subprotocols independently, it can be verified that not only does the overall protocol allocate all of the cake but at each point during the protocol, the allocated cake is envy-free. It can also be verified that the overall protocol is bounded by B' Robertson and Webb queries. The bound established is huge. On the positive side, the paper opens the door to new work on finding the optimal bound. Getting a clearer understanding of the complexity of envy-freeness is an interesting direction for future work.

Acknowledgments

The authors thank Sajid Aziz, Simina Brânzei, Omer Lev, Abdallah Saffidine, Erel Segal-Halevi, Jade Tan-Holmes, and all the reviewers of FOCS 2016 for comments. They also thank Steven Brams and Ariel Procaccia for pointers and support.

References

[1] H. Aziz and S. Mackenzie. A discrete and bounded envy-free cake cutting protocol for four agents. In *Proc. of 48th STOC*, pages 454–464. ACM Press, 2016.

[2] H. Aziz and S. Mackenzie. A discrete and bounded envy-free cake cutting protocol for any number of agents. Technical Report arXiv:1604.03655, arXiv.org, 2016.

[3] H. Aziz and C. Ye. Cake cutting algorithms for piecewise constant and piecewise uniform valuations. In *Proc. of 10th WINE*, pages 1–14, 2014.

[4] J. B. Barbanel. *The Geometry of Efficient Fair Division*. Cambridge University Press, 2005.

[5] J. B. Barbanel and S. 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.

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

[7] S. J. Brams and A. D. Taylor. *Fair Division: From Cake-Cutting to Dispute Resolution*. Cambridge University Press, 1996.

[8] S. J. Brams, A. D. Taylor, and W. S. Zwicker. A moving-knife solution to the four-person envy-free cake division. *Proceedings of the American Mathematical Society*, 125(2):547–554, 1997.

[9] Y. J. Cohler, J. K. Lai, D. C. Parkes, and A. D. Procaccia. Optimal envy-free cake cutting. In *Proc. of 25th AAAI Conference*, pages 626—631. AAAI Press, 2011.

[10] X. Deng, Q. Qi, and A. Saberi. Algorithmic solutions for envy-free cake cutting. *Mathematics of Operations Research*, 60(6): 1461–1476, 2012.

[11] J. Edmonds and K. Pruhs. Cake cutting really is not a piece of cake. In *Proc. of 17th SODA*, pages 271–278, 2006.

[12] G. Gamow and M. Stern. *Puzzle-math*. Macmillan, 1958.

[13] S. Garfunkel. For all practical purposes social choice. *COMAP*, 1988.

[14] C. Lindner and J. Rothe. Cake-cutting: Fair division of divisible goods. In J. Rothe, editor, *Economics and Computation: An Introduction to Algorithmic Game Theory, Computational Social Choice, and Fair Division*, chapter 7. Springer-Verlag, 2015.

[15] O. Pikhurko. On envy-free cake division. *The American Mathematical Monthly*, 107(8):736–738, 2000.

[16] A. D. Procaccia. Thou shalt covet thy neighbor's cake. In *Proc. of 21st IJCAI*, pages 239–244. AAAI Press, 2009.

[17] A. D. Procaccia. Cake cutting: Not just child's play. *Communications of the ACM*, 56(7):78–87, 2013.

[18] A. D. Procaccia. Cake cutting algorithms. In F. Brandt, V. Conitzer, U. Endriss, J. Lang, and A. D. Procaccia, editors, *Handbook of Computational Social Choice*, chapter 13. Cambridge University Press, 2016.

[19] J. M. Robertson and W. Webb. Near exact and envy-free cake division. *Ars Combinatorica*, 45:97–108, 1997.

[20] J. M. Robertson and W. A. Webb. *Cake Cutting Algorithms: Be Fair If You Can*. A. K. Peters, 1998.

[21] A. Saberi and Y. Wang. Cutting a cake for five people. In *Proc. of 5th International Conference on Algorithmic Aspects in Information and Management (AAIM)*, LNCS, pages 292–300. Springer, 2009.

[22] E. Segal-Halevi, A. Hassidim, and Y. Aumann. Waste makes haste: Bounded time protocols for envy-free cake cutting with free disposal. In *Proc. of 14th AAMAS Conference*, pages 901–908. IFAAMAS, 2015.

[23] H. Steinhaus. The problem of fair division. *Econometrica*, 16: 101–104, 1948.

[24] W. Stromquist. Envy-free cake divisions cannot be found by finite protocols. *The Electronic Journal of Combinatorics*, 15 (R11), 2008.

[25] F. E. Su. Rental harmony: Sperner's lemma in fair division. *American Mathematical Monthly*, 10:930–942., 1999.

[26] W. Thomson. Children crying at birthday parties. Why? *Economic Theory*, 31:501–521, 2007.