A General Framework for Fair Division

Galen Collins
12/09/12

I. Introduction

In observing the various algorithms proposed for dividing resources fairly among two or more parties, one question seemed to occur over and over again: how exactly does this map onto a real world problem. Ideally the algorithms would provide a framework for adapting their themselves to the various types of division in the real world, however they usually stop short at simple examples. In this paper I present cakery, a generic framework that eases the translation process of real world problems while still delivering the results each algorithm guarantees.

II. Description of Framework

Cakery is a general framework for describing resources and managing the preferences of one or more users to various pieces of the aforementioned resources. The basic idea is to force the library consumer to implement a minimal interface for their resource type which describe a few primitive operations. Once completed, these primitives can be composed to form higher level utilities that can be used to implement any proposed fair division algorithms in a generic and very high level fashion. In order to further discuss cakery, it is necessary to discuss the basic elements in a bit more detail.

As previously mentioned, cakery relies on the user to complete an interface pair for their resource type: one for operations on the resource and one for preferences on that resource. In order to complete the interfaces, the user must supply the following operations:

- **actual_value –** Returns a value that can be used to sort resources without having to involve user preferences. An example is the width of a slice (smaller vs larger).
- **clone –** Returns a shallow copy of this resource that can be used for destructive actions like each user proposing a trimming or a resource to meet the suggested value.
- **as_collection –** Returns a projection of this resource into a collection which is used for various auction based algorithms. An example is converting a tract of land into one acre plots for an auction.
- **remove –** Given a resource piece, removes that piece from the current resource. An example is a collection of items where an assigned item is removed from the lot.
- **append –** Given a resource piece, append it back into the current resource. An example would be to return a trimming of a cake back into the cake for further division.
- **find_piece –** Given a requested weight value, return a piece of the current resource if available that meets the required value. An example is asking a user to mark half of a cake in their view.
- **value_of –** Given a resource, return the value of said resource to the current user. An example

would be to ask a user what values they would give the various slices of a cake.

With these interfaces defined, it is now possible to define all the higher level primitives that can be used to write simple algorithms. Without explaining all the primitives, it is still valuable to see a few illustrations of implementing algorithms using them:

```
def divide_and_choose():
    slices = {}
    cutter, picker = randomize_items(users)
    pieces = create_equal_pieces(cutter, cake, 2)
    slices[picker] = choose_best_piece(picker, pieces)
    slices[cutter] = choose_and_remove(pieces)
    return slices
```

```
def sealed_bids_auction():
    slices = defaultdict(list)
    users  = randomize_items(users)
    for piece in cake.as_collection():
        cutter = choose_highest_bidder(users, piece)
        slices[cutter].append(piece)
    return slices
```

III. Explanation of Two Implementations

In creating cakery, the goal was to cover four basic resource types in order to demonstrate its generic nature[1]:

1. a generic continuous resource that allows approximate results
2. a specific continuous resource that allows for exact results
3. a discrete resource that does not allow elements to be repeated
4. a discrete resource that allows elements to be repeated zero or more times

Although all four have been implemented, this paper will simply focus on cases 2 and 3 which will be named Interval and Collection resources respectively. In describing how these two are used in the algorithms, it makes sense to simply describe how they were implemented. The first to be described is the interval resource:

- **actual_value –** This returns the total length of all the available interval ranges: *sum(end – start for start, end in intervals)*.
- **clone –** Since the intervals are represented as a list of range tuples *(start, end)*, a copy of the list is simply returned.
- **as_collection –** The interval range is simple split into a given count of pieces which can be specified by the user.
- **remove –** This will remove the supplied interval from the current range, possible leaving gaps in the range.
- **append –** This will append the supplied interval to the current range and then form a compaction process to attempt to minimize or remove any gaps in the range.
- **find_piece –** This uses a Stern Brocot tree to perform a binary fraction search with the supplied user value_of until the resulting fraction is within some given threshold of the requested value.
- **value_of –** This calculates the area over the supplied intervals. Since the value functions can only be linear, the implementation can simply use the integral of the linear equation to obtain exact results.

Continuing, the collection resource is defined as follows:

- **actual_value –** If the user has wrapped the items with the supplied item value wrapper which associates each item with a real world appraised value, the result is the sum of the item appraised values. Otherwise, it is simply the number of items in the collection.
- **clone –** Since the collection is represented as a list of items, a shallow copy of the list is returned.
- **as_collection –** The collection is simply returned as one new resource per each item in the collection.
- **remove –** This will remove the supplied item(s) from the collection.
- **append –** This will add the supplied item(s) to the collection.
- **find_piece –** This just tries every possible combination of items and returns the closest match to the requested weight.
- **value_of –** This performs a sum of the user supplied value for each item in the collection: *sum(value(item) for item in collection)*.

IV. Analysis of Results

In analyzing the results of cakery, I will only focus on a few algorithms along with the two aforementioned resource types. In order to demonstrate continuous and discrete resources both working on discrete and continuous algorithms, I will choose algorithms so that each group is represented: Divide and Choose, Dubins Spanier Moving Knife, Knaster Sealed Bids[2].

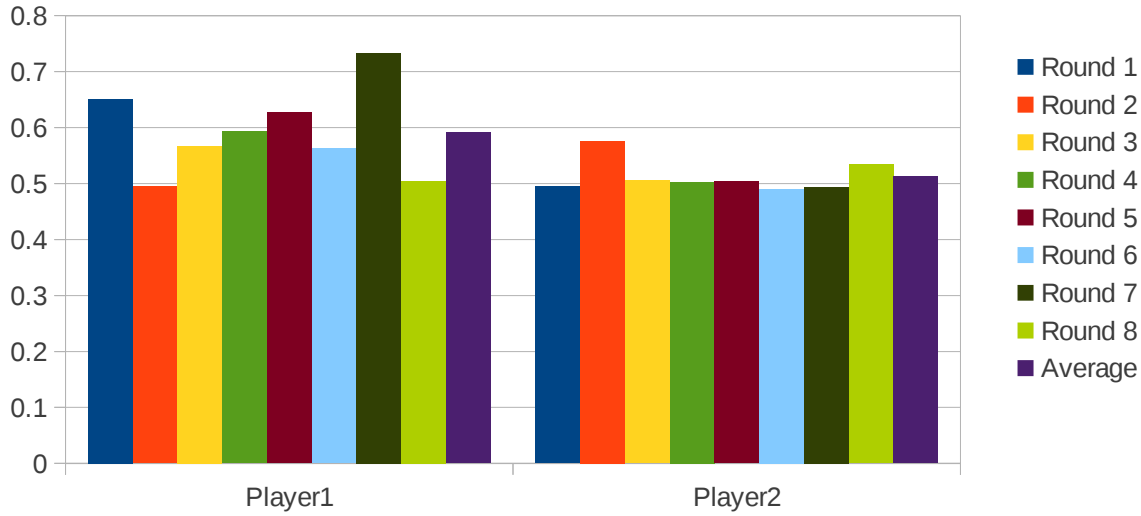
All three algorithms were able to meet their stated demands in terms of being envy-free, proportional,

and equitable regardless of which resource they were supplied with. It should be noted that the results are all within reason and the following stipulations will apply:

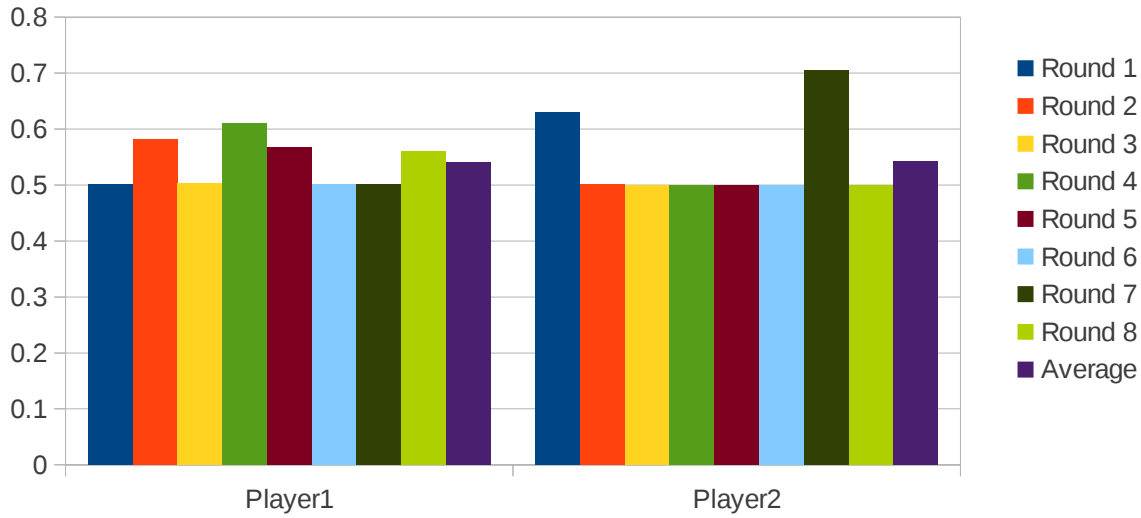
- If there is not enough collection items or items with necessary values, it becomes impossible using continuous division algorithms to achieve a fair division.
- In an attempt to make cakery as generic as possible (including being able to use int, float, or decimals), a number of the values are approximated which makes results correct to some threshold.
- All testing was performed using the random data generation functions for each resource method[1].

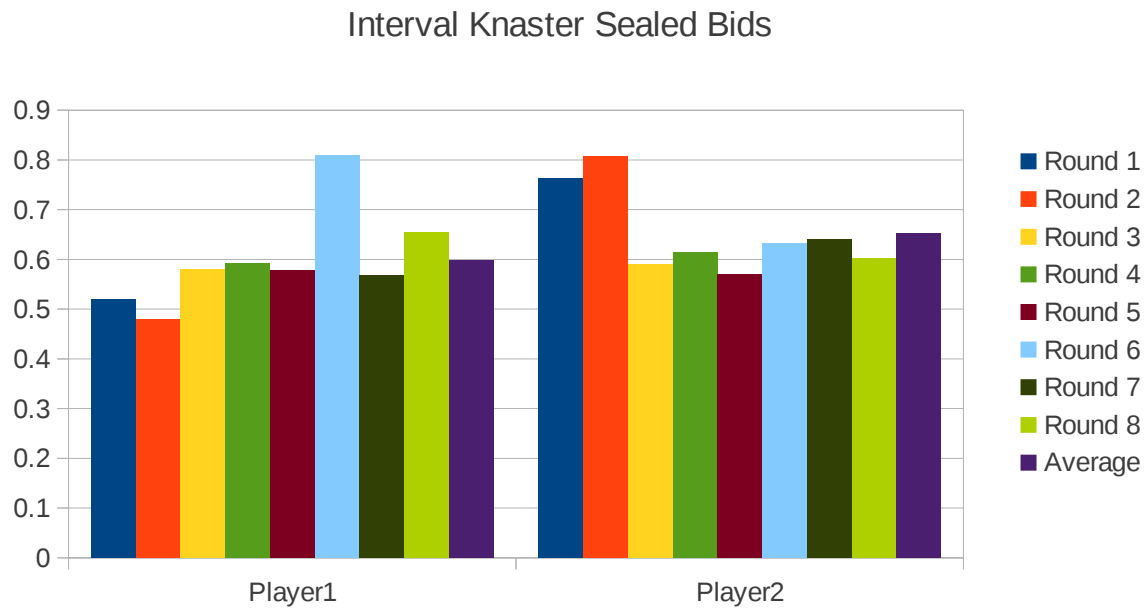
The results[3] that are to follow clearly demonstrate that each algorithm is generally envy free, roughly equitable, and generally proportional (strongly in a number of cases). It should also be noted that the Knaster Sealed Bids algorithm provides perfectly equitable and envy free results by suggesting a piece that can be divided between users. This was not applied in the results as it breaks down for indivisible items unless they are sold for the appraised value.

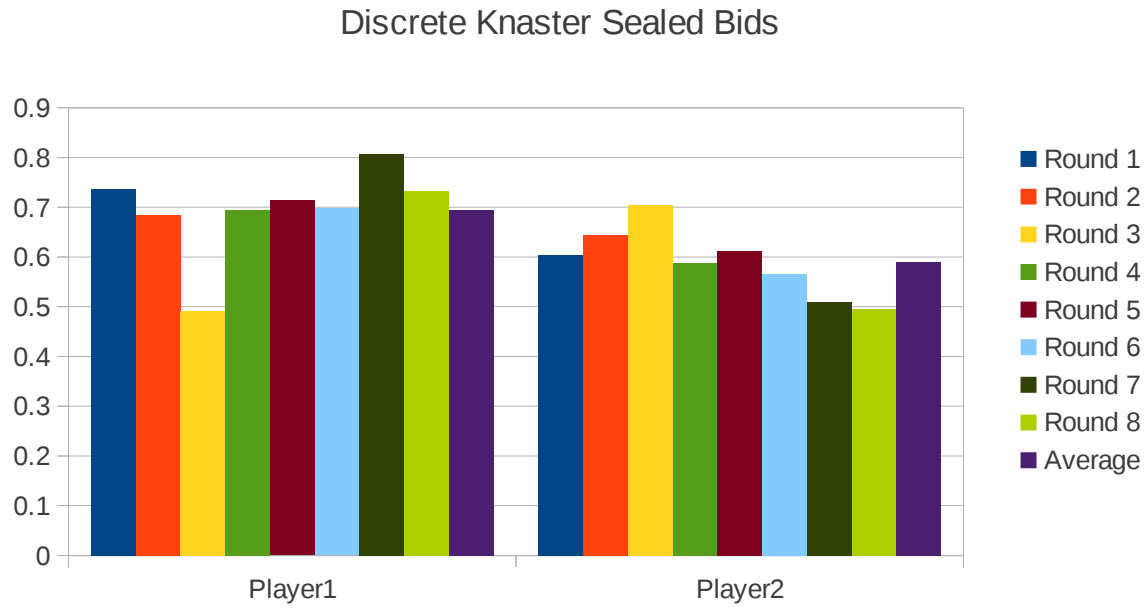
Interval Divide and Choose



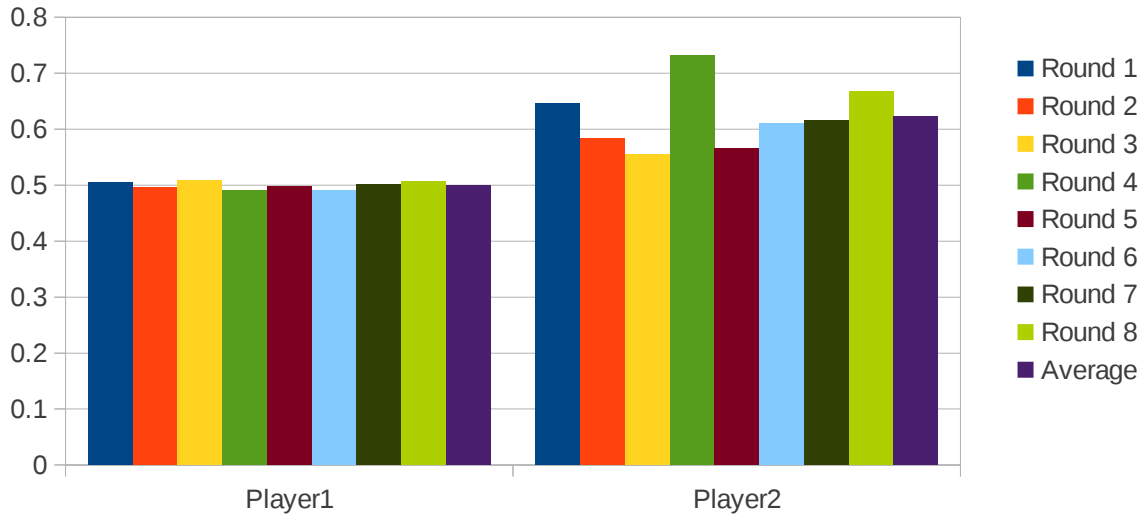
Discrete Divide and Choose



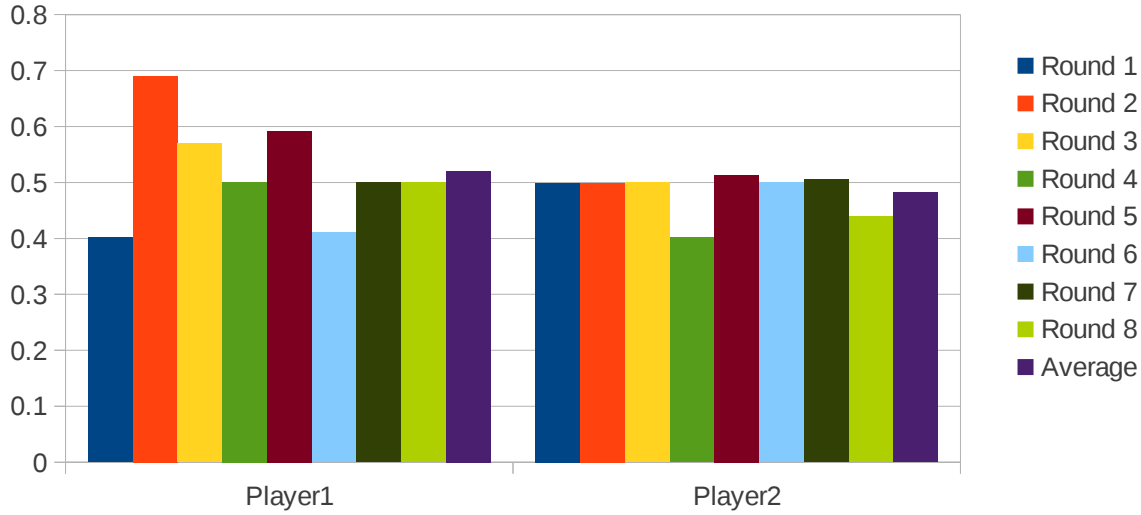

Interval Knaster Sealed Bids
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
Player1
Player2
Round 1
Round 2
Round 3
Round 4
Round 5
Round 6
Round 7
Round 8
Average


Discrete Knaster Sealed Bids
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
Player1
Player2
Round 1
Round 2
Round 3
Round 4
Round 5
Round 6
Round 7
Round 8
Average

Interval Dubins Spanier



Discrete Dubins Spanier



V. Conclusion

By examining the code required to implement each algorithm using the cakery framework versus implementations by other students in the class, it should be clear that cakery offers much cleaner, more understandable, and easier to reuse implementations. As demonstrated by the number of algorithms from the class that have been implemented in cakery, adapting new algorithms is trivial (many of which where implemented in minutes).

Continuing, the analytical results clearly indicate that the guarantees specified by the algorithms can be delivered for varying resources without having to specifically modify the code. It has been shown that approximately if not exact results can also be returned from algorithms that were never designed with other resource types in mind if solutions do indeed exist. Finally, the framework allows custom extensions to algorithms to suggest resolutions when perfect results cannot be achieved (as in the case in adjusted winners). Given all of the above arguments, cakery has been shown to be an ideal fair division algorithm playground for quickly testing new ideas while being able to correctly verify results.

VI. Footnotes

1. The described resource type as well as the remaining resource types can be viewed in the course provided source repository.
2. All of these algorithms as well a set of recognized data files for collection and interval resources are available on the course provided web page so that basic results can be repeated. As long as the formats are maintained, any other provided data files will work equally as well.
3. The resulting raw data is attached at the end of this report.

VII. Raw Test Data

Discrete Divide And Choose

Round 1	Round 2	Round 3	Round 4	Round 5	Round 6	Round 7	Round 8	Average
0.5003208999	0.5813492479	0.5022468631	0.6110394163	0.5675630934	0.5008907879	0.5003319046	0.5598554213	0.5404497043
0.6291409093	0.5006333191	0.4996498074	0.4998057493	0.4998999568	0.4999734886	0.7042058605	0.4998911966	0.541650036

Interval Divide And Choose

Round 1	Round 2	Round 3	Round 4	Round 5	Round 6	Round 7	Round 8	Average
0.6508067776	0.4945740319	0.5675746577	0.5928493026	0.6267785366	0.5630084119	0.7331245401	0.50424574	0.5916202498
0.4948172822	0.5756631281	0.5058946064	0.5030680555	0.5038306224	0.4905607736	0.4936079497	0.5336260111	0.5126335536

Discrete Knaster Sealed Bids

Round 1	Round 2	Round 3	Round 4	Round 5	Round 6	Round 7	Round 8	Average
0.7358344259	0.6843372922	0.4903966991	0.6942845174	0.7135134643	0.6973887866	0.8063069415	0.731561699	0.6942029783
0.6030900139	0.6432622	0.7043155332	0.5871422974	0.6119960613	0.5664353469	0.5101344681	0.494690143	0.590133258

Interval Knaster Sealed Bids

Round 1	Round 2	Round 3	Round 4	Round 5	Round 6	Round 7	Round 8	Average
0.5195578516	0.4805177848	0.5804019461	0.5929694376	0.5790262643	0.8101279336	0.5683187377	0.654657815	0.5981972213
0.7631767713	0.8067306618	0.5890153759	0.6140087707	0.5705338374	0.6324082916	0.6413533463	0.6027079462	0.6524918752

Discrete Dubins Spanier

Round 1	Round 2	Round 3	Round 4	Round 5	Round 6	Round 7	Round 8	Average
0.4018863523	0.6899057906	0.5700965259	0.5006343565	0.5920480638	0.410668825	0.5001019997	0.5000329219	0.5206718545
0.4995607151	0.4989658016	0.4997329293	0.4012097391	0.5131855783	0.5003026032	0.5049595656	0.4391044532	0.4821276732

Interval Dubins Spanier

Round 1	Round 2	Round 3	Round 4	Round 5	Round 6	Round 7	Round 8	Average
0.5047060009	0.4952854055	0.5080328371	0.4911742091	0.4974577402	0.4903254205	0.5005060504	0.5061690332	0.4992070871
0.6464009274	0.5843181187	0.5554942365	0.7315439858	0.5646721369	0.6098099878	0.6154849909	0.6679947906	0.6219648968