Algorithms for Matching and Fair Division

Jan van Eijck
CWI & ILLC, Amsterdam

ESSLLI, Opole, August 10, 2012

Abstract

This lecture deals with algorithms for matching and assignment, and for fair division.

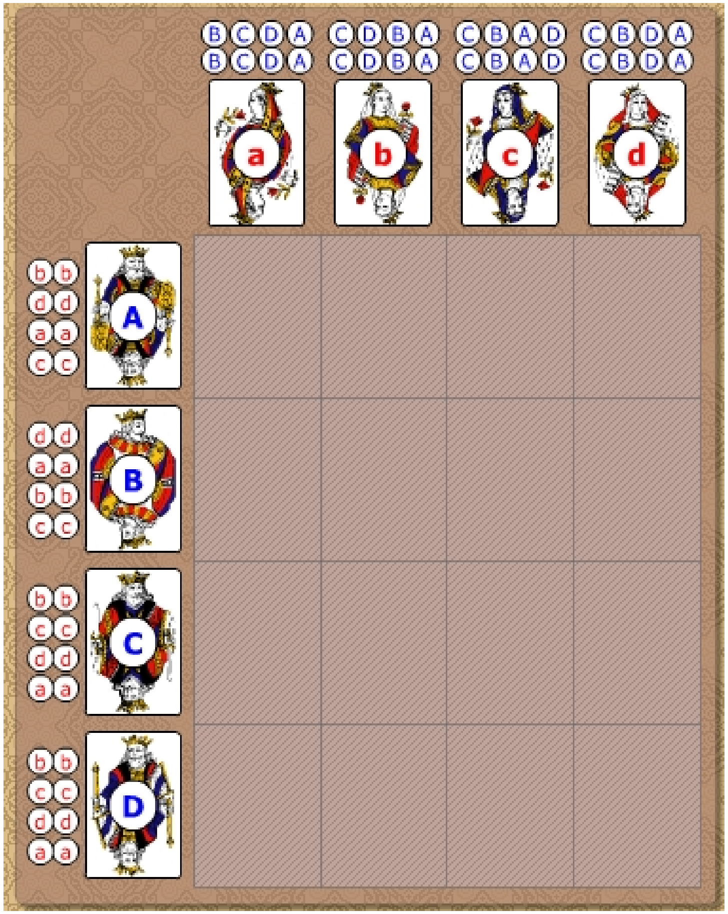
The matching problem is the problem of connecting nodes in a bipartite graph, while making sure that certain requirements are met. An assignment problem is a problem of assigning tasks to agents. Any agent can be assigned to perform any task, incurring some cost that may vary depending on the agent-task assignment. The requirement is to perform all tasks by assigning distribing the tasks over the agents in such a way that the total cost of the assignment is minimized.

The problem of cutting a cake among N participants in such a way that every participant is satisfied with his or her share will serve as an exemplar for fair division.

```
module MatchingDivision

where

import List
import While
import Assert
--import Reasoning (update,updates)
```

The Stable Marriage Problem

Suppose equal sets of men and women are given, each man has listed the women in order of preference, and vice versa for the women. A stable marriage match between men and women is a one-to-one mapping between the men and women with the property that if a man prefers another woman over his own wife then that woman does not prefer him to her own husband, and if a woman prefers another man over her own husband, then that man does not prefer her to his own wife.


B C D A
B C D A
C D B A
C D B A
C B A D
C B A D
C B D A
C B D A
a
b
c
d
b b
d d
a a
c c
A
d d
a a
b b
c c
B
b b
c c
d d
a a
C
b b
c c
d d
a a
D

The computer scientists Gale and Shapley proved that stable matchings always exist, and gave an algorithm for finding such matchings, the so-called Gale-Shapley algorithm [7]. This has many important applications, also outside of the area of marriage counseling.

See `http://en.wikipedia.org/wiki/Stable_marriage_problem` for more information.

Gale-Shapley

Gale-Shapley algorithm for stable marriage

```
1. Initialize all $m$ in $M$ and $w$ in $W$ to free;
2. while there is a free man $m$ who still has a woman $w$ to propose to
    (a) $w$ is the highest ranked woman to whom $m$ has not yet proposed
    (b) if $w$ is free, $(w, m)$ become engaged
        else (some pair $(w, m')$ already exists)
        if $w$ prefers $m$ to $m'$
           i. $(w, m)$ become engaged
          ii. $m'$ becomes free
        else $(w, m')$ remain engaged
```

On the next page is a slightly more formal version using named variables.

Gale-Shapley algorithm for stable marriage (ii)

```
1. Let equal-sized sets $M$ and $W$ of men and women be given.
   Let pr be a preference function.
2. $F := M$, $E := \emptyset$.
3. While $F \neq \emptyset$ do
   (a) take $m \in F$;
   (b) $w :=$ the highest ranked woman to whom $m$ has not yet proposed;
   (c) delete $w$ from the preference list of $m$;
   (d) if $\neg\exists m' : (w, m') \in E$ then
         i. $E := E \cup \{(w, m)\}$;
        ii. $F := F - \{m\}$;
       else ($\exists m' : (w, m') \in E$) if $\mathrm{pr}_w m m'$ then do
         i. $E := (E - \{(w, m'\}) \cup \{(w, m)\}$;
        ii. $F := (F - \{m\}) \cup \{m'\}$;
```

Parameters of the While Loop

- the preference list of the men
- the list of current engagements
- the list of current free men.

It may look like we also need the preference list of the women as a parameter, but this list is not used for stating the exit condition and it is not changed by the step function, and we can keep it outside the loop.

So it is possible to phrase the algorithm in terms of while3.

Type Declarations

Type declarations, for readability of the code:

```
type Man    = Int
type Woman = Int
type Mpref = [(Man,[Woman])]
type Wpref = [(Woman,[Man])]
type Engaged = [(Woman,Man)]
```

Example tables of preferences

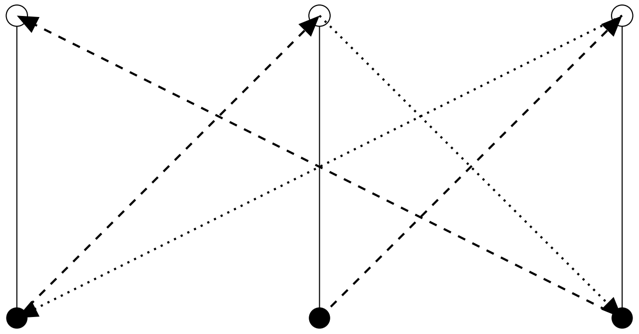
```
mt :: Mpref
mt = [(1,[2,1,3]), (2, [3,2,1]), (3,[1,3,2])]

wt :: Wpref
wt = [(1,[1,2,3]),(2,[3,2,1]), (3,[1,3,2])]
```

In `mt`, the entry `(1,[2,1,3])` indicates that man 1 prefers woman 2 over woman 1 and woman 1 over woman 3. We assume that preferences are transitive, so man 1 also prefers woman 2 over woman 3.

Stable Match?

```
mt = [(1,[2,1,3]), (2, [3,2,1]), (3,[1,3,2])]
wt = [(1,[1,2,3]),(2,[3,2,1]), (3,[1,3,2])]
```

○ for the women, ● for the men:



Conversion

An auxiliary function for converting a preference list to a function, which allows us to express w prefers m to m’ in a simple way.

```
type PrefFct = Int -> Int -> Int -> Bool

plist2pfct :: [(Int,[Int])] -> PrefFct
plist2pfct table x y y' =
  let
    Just prefs = lookup x table
  in elem y (takeWhile (/= y') prefs)
```

Initialisation

The list of all men is extracted from the table of men's preferences, all men are free, no-one is engaged to start with. Note that `map fst mpref` gives us the list of all men.

```
stableMatch :: Wpref -> Mpref -> Engaged
stableMatch wpref mpref =
  let
     men     = map fst mpref
     free    = men
     engaged = []
  in
  stable wpref mpref free engaged
```

While Loop

The test function for the while loop just checks whether the list of free men is exhausted.

The step function for the while loop has an argument of type `Mpref` for the current list of men's preferences, `Engaged` for the list of currently engaged (w, m) pairs, and an argument of type `[Man]` for the list of currently free (not engaged) men. The list of men's preferences changes in the loop step, for each woman that a man proposes to is crossed from his preference list.

The algorithm assumes that there are equal numbers of men and women to start with, and that both men and women have listed all members of the opposite sex in order of preference.

Implementation

```
stable ::  Wpref -> Mpref -> [Man] -> Engaged
       -> Engaged
stable wpref = let
   wpr = plist2pfct wpref
 in while3 (\ _ free _ -> not (null free))
           (\ mpr (m:free) engaged  ->
            let
               Just (w:ws) = lookup m mpr
               match = lookup w engaged
               mpr' = (m,ws) : (delete (m,w:ws) mpr)
               (engaged',free') = case match of
                   Nothing -> ((w,m):engaged,free)
                   Just m' ->
                     if wpr w m m' then (
                         (w,m) :
                           (delete (w,m') engaged),
                          m':free)
                     else (engaged, m:free)
            in (mpr',free',engaged'))
```

Recall the example tables of preferences

```
mt :: Mpref
mt = [(1,[2,1,3]), (2, [3,2,1]), (3,[1,3,2])]

wt :: Wpref
wt = [(1,[1,2,3]),(2,[3,2,1]), (3,[1,3,2])]
```

A simple example

To demonstrate how the function `stableMatch` is used:

```
makeMatch = stableMatch mt wt
```

This gives:

```
*AS> makeMatch
[(2,2),(3,3),(1,1)]
```

This gives (w, m) pairs. So woman 1 is married to man 1, and so on. Note that the first woman ends up with the man of her preference, but the other two women do not. But this match is still stable, for although the second woman is willing to swap her husband for the third man, she is at the bottom of his list. And so on.

Correctness: Termination

To show that this algorithm is correct, we have to show two things: termination and stability of the constructed match.

Exercise 1 Show that if there are equal numbers of men and women, the algorithm always terminates. Hint: analyze what happens to the preference lists of the men. Observe that no man proposes to the same woman twice.

See exercises for today.

Correctness: Stability

To show stability, we have to show that each step through the step function preserves the invariant “the set of currently engaged men and women is stable.”

What does it mean for E to be stable on W and M? Let us use $\text{pr}_w mm'$ for w prefers m over m'.

- $\forall(w, m) \in E \forall(w', m') \in E$: if $\text{pr}_m w'w$ then $\text{pr}_{w'} m'm$;
- $\forall(w, m) \in E \forall(w', m') \in E$: if $\text{pr}_w mm$ then $\text{pr}_{m'} w'w$.

What is the requirement on `free`?

- free equals the set of all men minus the men that are engaged.

We see that these requirements hold for the first call to `stable`, for in that call `engaged` is set to $[]$ and `free` is set to the list of all men. The empty list of engaged pairs is stable by definition.

Inspection of the Step Function

Next, inspect what happens in the step function for `stable`. The precondition for the step to be performed is that there is at least one free man m left. m proposes to the woman w who is on the top of his list.

Proposal to a Free Woman

If w is free, w accepts the proposal, and they become engaged. Is the new list of engaged pairs stable? We only have to check for the new pair (w, m).

- Suppose that there is a free w' with $\text{pr}_m w' w$. This cannot be, for w is at the top of m's list.
- Suppose there is m' with $\text{pr}_w m' m$. Then if m' is engaged, this must mean that not $\text{pr}_{m'} w w'$, where w' is the fiancee of m'. For otherwise m' would have proposed to w instead of to w'.
- The new list of free men equals the old list, minus m. This is correct, for m just got engaged.

Proposal to an Engaged Woman

Now the other case: w is already engaged. There are two subcases. In case w prefers her own current fiancee, nothing happens. The resulting list of engaged pairs is still stable. The list of free men remains the same, for m proposed and got rejected.

In case w prefers m to her own fiancee m', she swaps: (w, m) replaces (w, m') in the list of engaged pairs. Again, it is easy to see that the resulting list of engaged pairs is stable. m gets replaced by m' on the list of free men.

So the two stability requirements are satisfied.

The requirement that in any call to `stable` its last argument contains the list of currently free men is also satisfied, for m gets replaced by m' on the list of free men.

An Assertive Version of the Stable Marriage Algorithm

An alternative to reasoning about the correctness of an algorithm is specification-based testing, which we will explore now.

One of the properties that has to be maintained through the loop step function for stable marriage is: “Each man is either free or engaged, but never both.” This is implemented as follows:

```
freeProp :: Mpref -> [Man] -> Engaged -> Bool
freeProp mpref free engaged = let
    men  = map fst mpref
    emen = map snd engaged
  in forall men (\x -> elem x free == notElem x emen)
```

Stability Property

The other invariant is the stability property. Here is the definition of stability for a relation E consisting of engaged (w, m) pairs:

$$\forall (w,m) \in E \forall (w',m') \in E$$
$$((\mathrm{pr}_w m'm \to \mathrm{pr}_{m'} w'w) \wedge (\mathrm{pr}_m w'w \to \mathrm{pr}_{w'} m'm)).$$

What this says (once more) is: if w prefers another guy m' to her own fiancee m then m' does prefer his own fiancee w' to w, and if m prefers another woman w' to his own fiancee w then w' does prefer her own fiancee m' to m.

Implementation

Once it is written like this it is straightforward to implement it, for we have all the ingredients:

```
isStable :: Wpref -> Mpref -> Engaged -> Bool
isStable wpref mpref engaged = let
    wf = plist2pfct wpref
    mf = plist2pfct mpref
  in
    forall engaged (\ (w,m) -> forall engaged
          (\ (w',m') -> (wf w m' m ==> mf m' w' w)
                         &&
                        (mf m w' w ==> wf w' m' m)))
```

Assertive Version of Stable Matching

This property can be used as a test on the output of `stableMatch`, as follows:

```
stableMatch' :: Wpref -> Mpref -> Engaged
stableMatch' = assert2 isStable stableMatch
```

Another Possibility

Another possibility is to use the assertion as an invariant. This version has to check stability for a new pair:

```
stablePair :: Wpref -> Mpref
           -> (Woman,Man) -> Engaged -> Bool
stablePair wpref mpref (w,m) engaged = let
    wf = plist2pfct wpref
    mf = plist2pfct mpref
  in
    forall engaged
          (\ (w',m') -> (wf w m' m ==> mf m' w' w)
                         &&
                        (mf m w' w ==> wf w' m' m))
```

Still Stable?

Distinguish between the full preference list of the men and their current preference list.

Use the current preference list to pick the current most preferred woman of the first free man. This excludes the women he has been rejected by, or who have swapped him for a better match in the preceding steps of the algorithm.

Check that in case the current most preferred woman of the first free man is still free, the result of their engagement does not spoil stability:

```
stillStable :: Wpref -> Mpref
            -> Mpref -> [Man] -> Engaged -> Bool
stillStable wpr mpr currentmpr [] eng = True
stillStable wpr mpr currentmpr (m:free) eng = let
   Just (w:ws) = lookup m currentmpr
   match = lookup w eng
 in
   match == Nothing ==> stablePair wpr mpr (w,m) eng
```

Version of the code including the two invariants

```
stableMatchA :: Wpref -> Mpref -> Engaged
stableMatchA wpref mpref =
  let
     men     = map fst mpref
     free    = men
     engaged = []
  in
  stableA wpref mpref mpref free engaged
```

We have one more parameter, for the preference function for the men. This is needed to implement the stability check correctly. The code inside the `while3` loop does not change.

```
stableA ::  Wpref -> Mpref ->
            Mpref -> [Man] -> Engaged -> Engaged
stableA wpref mpref = let
    wf     = plist2pfct wpref
 in while3 (\ _ free _ -> not (null free))
  (invar3 freeProp
  (invar3 (stillStable wpref mpref)
  (\ mpr (m:free) engaged  ->
      let
        Just (w:ws) = lookup m mpr
        match = lookup w engaged
        mpr' = (m,ws) : (delete (m,w:ws) mpr)
        (engaged',free') = case match of
          Nothing -> ((w,m):engaged,free)
          Just m' ->
            if wf w m m' then (
                  (w,m) : (delete (w,m') engaged),
                   m':free)
            else (engaged, m:free)
      in (mpr',free',engaged'))))
```

Note that we have stacked the two invariant tests, by wrapping one invariant around the wrap of the other invariant around the step function.

Example preference list for the men

```
mt2 = [(1, [1, 5, 3, 9, 10, 4, 6, 2, 8, 7]),
       (2, [3, 8, 1, 4, 5, 6, 2, 10, 9, 7]),
       (3, [8, 5, 1, 4, 2, 6, 9, 7, 3, 10]),
       (4, [9, 6, 4, 7, 8, 5, 10, 2, 3, 1]),
       (5, [10, 4, 2, 3, 6, 5, 1, 9, 8, 7]),
       (6, [2, 1, 4, 7, 5, 9, 3, 10, 8, 6]),
       (7, [7, 5, 9, 2, 3, 1, 4, 8, 10, 6]),
       (8, [1, 5, 8, 6, 9, 3, 10, 2, 7, 4]),
       (9, [8, 3, 4, 7, 2, 1, 6, 9, 10, 5]),
       (10, [1, 6, 10, 7, 5, 2, 4, 3, 9, 8])]
```

And for the women

```
wt2 =[(1, [2, 6, 10, 7, 9, 1, 4, 5, 3, 8]),
      (2, [2, 1, 3, 6, 7, 4, 9, 5, 10, 8]),
      (3, [6, 2, 5, 7, 8, 3, 9, 1, 4, 10]),
      (4, [6, 10, 3, 1, 9, 8, 7, 4, 2, 5]),
      (5, [10, 8, 6, 4, 1, 7, 3, 5, 9, 2]),
      (6, [2, 1, 5, 9, 10, 4, 6, 7, 3, 8]),
      (7, [10, 7, 8, 6, 2, 1, 3, 5, 4, 9]),
      (8, [7, 10, 2, 1, 9, 4, 8, 5, 3, 6]),
      (9, [9, 3, 8, 7, 6, 2, 1, 5, 10, 4]),
      (10, [5, 8, 7, 1, 2, 10, 3, 9, 6, 4])]
```

And a test run with this

```
makeMatch2 = stableMatchA mt2 wt2
```

This gives:

```
*AS> makeMatch2
[(4,6),(5,10),(1,9),(9,8),(7,7),(8,5),
 (10,1),(6,2),(3,4),(2,3)]
```

The Weighted Matching Problem

The weighted matching problem seeks to find a matching in a weighted bipartite graph that has maximum weight. Maximum weighted matchings do not have to be stable, but in some applications a maximum weighted matching is better than a stable one.

College or Hospital Admission

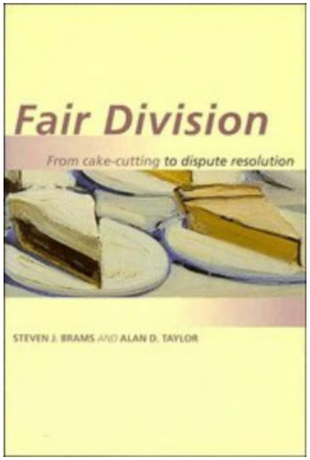
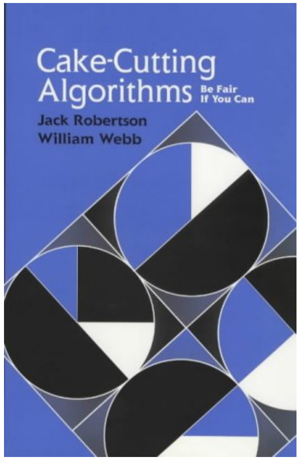
In the college/hospital admission problem, several students or doctors can propose to the same college or hospital.

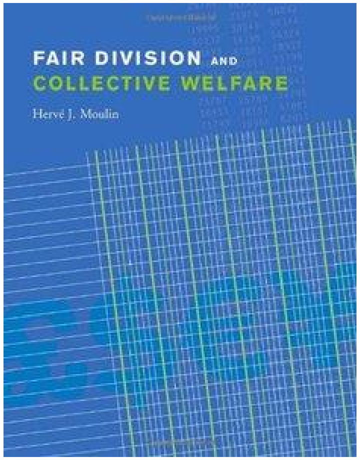
The Task Assignment Problem

In the task assignment problem, a number of tasks has to be performed by a number of agents at minimal cost. An example is a taxi cab scheduling systems, where a number of taxis at different locations have to pick up a number of customers at different locations.

Fair Division

For background on fair division, see [2, 3, 1, 10, 12].


Fair Division
From cake-cutting to dispute resolution
STEVEN J. BRAMS AND ALAN D. TAYLOR




A Tale From India

Two farmers, Ram and Shyam were eating chapatis. Ram had 3 pieces of the flat, round bread and Shyam had 5. A traveller who looked hungry and tired rode up to the two men. Ram and Shyam decided to share their chapatis with him. The 3 men stacked the 8 chapatis (like pancakes) and cut the stack into 3 equal parts. They shared the pieces equally and ate until nothing was left. The traveller, who was a nobleman, was so grateful that he gave the two farmers 8 gold coins for his share of the food.

After the traveller left, Ram and Shyam wondered how they should share the 8 gold coins. Ram said that there were 8 coins and only 2 people, so each person should get an equal share of 4 coins. “But thats not fair,” said Shyam, “since I had 5 chapatis to begin with.” Ram could see his point, but he didnt really want to give 5 of the coins to Shyam. So he suggested they go see Maulvi, who was very wise. Shyam agreed.

Ram and Shyam told the whole story to Maulvi. After thinking for a long time, he said that the fairest way to share the coins was to give Shyam 7 coins and Ram only 1 coin. Both men were surprised. But when they asked Maulvi to explain his reasoning, they were satisfied that it was a fair division of the 8 coins.

T.V. Padma, Mathematwist: Number Tales from Around the World [11]

One could easily conclude from this tale that fairness is too subjective to allow for formal analysis. We will demonstrate in this chapter that that would be a mistake.

See Exercises for Further Reflection

Cake Cutting Algorithms

Problems of fair division can often be represented as cake cutting problems. The assumption is that the cake is non-heterogeneous: it contains a variety of goodies (cream topping, fruit, chocolate, etcetera) at a variety of places, and these goodies are valued differently by the participants in the cake-sharing process. Without this assumption cake-cutting is easy, of course. Just give each of N contestants $\frac{1}{N}$ of the cake, with $\frac{1}{N}$ calculated according to the valuation they all agree on.

Assumption of Selfishness

We will assume the participants in a cake cutting process are unashamedly selfish. Each wants the largest piece he can get away with. Here is a relevant quote from [12]:

> We encourage the reader to envision yourself in a life-or-death struggle with siblings to see that you get a "fair share" of a literal leftover piece of your favorite cake. Don't give up a crumb!

In case a pie is to be shared between two people, we can use the oldest social procedure of the world. Cut and choose (also known as 'I cut, you choose') is a procedure for two-person fair division of some desirable or undesirable heterogeneous good.

Valuations

Indeed, let X be a set representing the good to be divided. A valuation function μ for X is a function from $\mathcal{P}(X)$ to $[0, 1]$ with the properties that

- $\mu(\emptyset) = 0$,
- $\mu(X) = 1$, and
- $A \subseteq B \subseteq X$ implies $\mu(A) \leq \mu(B)$.

If two valuation measures μ_m and μ_y (for my valuation and your valuation of X) are different, this means that you and I value some items in X differently.

I Cut, You Choose

Cake cutting for two: "I Cut, You Choose"

1. Let X be given, together with two valuation functions $\mu_1, \mu_2 : \mathcal{P}(X) \to [0..1]$.
2. Person 1 uses μ_1 to divide X into X_1 and X_2 with $\mu_1(X_1) = \mu_1(X_2)$.
3. If $\mu_2(X_1) \geq \mu_2(X_2)$, person 2 gets X_1, otherwise person 2 gets X_2.

If the two participants have different value judgements on parts of the goods, it is possible that both participants feel they received more than 50 percent of the goods. It follows, as was already observed by Hugo Steinhaus in 1948, that there exists a division that gives both parties more than their due part; "this fact disproves the common opinion that differences in estimation make fair division difficult"[13].

Knowledge Matters

It matters whether the valuations are known to the other party. Such knowledge can be used to advantage by the one who cuts. First consider the case that your valuation is unknown to me, and vice versa. Then if I cut, the best I can do is follow the algorithm above: pick sets $A, B \subseteq X$ with $A \cap B = \emptyset$, $A \cup B = X$, and $\mu_m(A) = \mu_m(B)$. If you choose, you will use μ_y to pick the maximum of $\{\mu_y(A), \mu_y(B)\}$. It follows immediately that cutting guarantees a fair share, but no more than that, while choosing holds a promise for a better deal. So if you ever get the choice between cutting and choosing in a situation where both parties only know their own valuation, then it is to your advantage to leave the cutting to the other person.

Moving Knife Algorithm

Below, to keep matters simple, we will assume that valuations are private (not known to the other participants). Here is a specification for the “Moving Knife” cake cutting algorithm for N participants, with $N \geq 2$ (described in [5]).

‘Moving knife’ algorithm for cake cutting

A knife is slowly moved at constant speed parallel to itself over the top of the cake. At each instant the knife is poised so that it could cut a unique slice of the cake. As time goes by the potential slice increases monotonely from nothing until it becomes the entire cake. The first person to indicate satisfaction with the slice then determined by the position of the knife receives that slice and is eliminated from further distribution of the cake. (If two or more participants simultaneously indicate satisfaction with the slice, it is given to ny one of them.) The process is repeated with the other $N - 1$ participants and with what remains of the cake [5].

Modelling Assumptions

To get a bit closer to implementation, assume that X is the line segment from 0 to 1. A valuation function for this is a function μ that assigns values to sub-intervals $[a_i \dots b_i]$ with $0 \leq a_i < b_i \leq 1$, with the following properties:

1. If I, J are subintervals of $[0..1]$ with $I \subseteq J$, then $\mu(I) \leq \mu(J)$,
2. $\mu([0..1]) = 1$, $\mu(\emptyset) = 0$,
3. if $0 \leq a < b \leq 1$, then $\mu[a..b] > 0$.

If a participant i holds valuation function μ_i, then $\text{Eval}(i, a, b) := \mu_i[a..b]$ expresses how i values the interval $[a..b]$.

Another useful function is $\text{Cut}(i, a, v)$, which gives the smallest $b > a$ for which $\mu_i[a..b] \geq v$.

Banach-Knaster Algorithm

Here is an algorithm due to the Polish mathematicians Banach and Knaster (described in [13]) using these concepts.

Banach-Knaster algorithm for cake cutting

1. Let N be the set of people sharing interval $[a..1]$.
2. If $N = 1$, the remaining person gets $[a..1]$.
3. Otherwise, let $i \in N$ be such that $b = \text{Cut}(i, a, \frac{1-a}{|N|})$ is minimal.
4. Give $[a..b]$ to i, and continue the cake cutting with $N - \{i\}$ and interval $[b..1]$.

Note the definition of $b = \text{Cut}(i, a, \frac{1-a}{|N|})$. This ensures that, in the valuation of i, the portion $[a..b]$ of the cake is worth $\frac{1}{|N|}$ of the current piece under discussion, which is $[a..1]$.

Trimming

This is often called Banach and Knaster's trimming algorithm, because one might think of the determination of

> "let $i \in N$ be such that $b = \text{Cut}(i, a, \frac{1-a}{|N|})$ is minimal"

as a process of trimming a given piece of cake down until no participant considers it too big anymore. Another way to think about this is as a process of determining where to cut, taking the valuations of all participants into account. (Looked at this way, the algorithm proposes a single cut, not a succession of cuts that get closer and closer to the final decision.)

Type Definitions

Here are some useful type abbreviations for an implementation:

```
type Agent     = Int
type Value     = Rational
type Boundary  = Rational
type Valuation = Agent -> Boundary -> Value
```

Evaluation Function

The evaluation function can be defined from a valuation table:

```
eval :: Valuation -> Agent
     -> Boundary -> Boundary -> Value
eval f i a b = (f i) b - (f i) a
```

Illustration

To illustrate this, it is useful to define an example valuation table. `table i b` gives the value that agent i assigns to the interval $[0..b]$. Agents 1 and 2 have uniform valuation: all parts of the cake are alike to them.Agent 3 and 4 values the first half more than the second half. With agent 5 and 6 it is the reverse. Agents 7 and 8 value the first one third of the cake less than the rest.

```
table :: Valuation
table 1 b = b
table 2 b = b
table 3 b = if b < 1/2
            then b * (3/2)
            else (3/4) + (1/2)*(b - 1/2)
table 4 b = if b < 1/2
            then b * (3/2)
            else (3/4) + (1/2)*(b - 1/2)
table 5 b = if b < 1/2
            then b * (1/2)
            else (1/4) + (3/2)*(b - 1/2)
table 6 b = if b < 1/2
            then b * (1/2)
            else (1/4) + (3/2)*(b - 1/2)
table 7 b = if b < 1/3
            then b * (1/2)
            else (1/6) + (5/4)*(b - 1/3)
table 8 b = if b < 1/3
            then b * (1/2)
            else (1/6) + (5/4)*(b - 1/3)
```

Next, some weird valuations:

```
table 9 b = let
    n = fromIntegral
        (length [ x | x <- goodies, x < b  ])
  in  n/3
table 10 b = if chocolate <= b then 1 else 0
```

This uses:

```
goodies = [ 1/3, 1/2, 2/3 ]
chocolate = 1/2
```

Agent 9 cares only about the goodies, while agent 10 is obsessed with chocolate: she cares about nothing else. Note that the valuations for these agents are not proper: they do not satisfy the condition that each non-empty segment of the cake should have a positive value.

Checks

Check that all valuation functions defined so far in this table are proper valuation functions.

Here is a partial check:

```
check :: Valuation -> Agent -> [Value]
check table agent =
  [ eval table agent 0 (1/fromIntegral(n)) |
                                     n <- [1..] ]
```

This gives:

```
*MatchingDivision> take 10 (check table 1)
[1 % 1,1 % 2,1 % 3,1 % 4,1 % 5,1 % 6,1 % 7,
 1 % 8,1 % 9,1 % 10]
*MatchingDivision> take 10 (check table 3)
[1 % 1,3 % 4,1 % 2,3 % 8,3 % 10,1 % 4,3 % 14,
 3 % 16,1 % 6,3 % 20]
```

```
*MatchingDivision> take 10 (check table 5)
[1 % 1,1 % 4,1 % 6,1 % 8,1 % 10,1 % 12,1 % 14,
 1 % 16,1 % 18,1 % 20]
*MatchingDivision> take 10 (check table 7)
[1 % 1,3 % 8,1 % 6,1 % 8,1 % 10,1 % 12,1 % 14,
 1 % 16,1 % 18,1 % 20]
```

The tables of agents 9 and 10 are different, for they contain intervals that are worth nothing at all, according to the agent:

```
*MatchingDivision> take 10 (check table 9)
[1 % 1,1 % 3,0 % 1,0 % 1,0 % 1,0 % 1,0 % 1,
 0 % 1,0 % 1,0 % 1]
*MatchingDivision> take 10 (check table 10)
[1 % 1,1 % 1,0 % 1,0 % 1,0 % 1,0 % 1,0 % 1,
 0 % 1,0 % 1,0 % 1]
```

We will use these weird valuations below to illustrate that the fairness of the Banach-Knaster algorithm depends on properties of the valuation functions.

Cut

The cut function can be defined from a valuation table provided we allow for some margin of error. The reason for this is that for cutting we need the inverse of the valuation, to determine the cut that yields a particular value.

```
epsilon = 1/10000

cut :: Valuation -> Agent
    -> Boundary -> Value -> Boundary
cut f i a v = approxCut f i v epsilon a a 1
```

Right cut location

The right cut location is found by binary search:

```
approxCut :: Valuation -> Agent -> Value -> Boundary
    -> Boundary ->  Boundary -> Boundary -> Boundary
approxCut f i v error a left right = let
   m = left + (right - left)/2
   guess = eval f i a m
 in
   if guess == v || right - left < error then m
   else if guess > v then
     approxCut f i v error a left m
   else approxCut f i v error a m right
```

Example cuts

```
*FDA> cut table 3 (1/2) (1/8)
3 % 4
*FDA> cut table 1 (1/2) (1/8)
5 % 8
*FDA> cut table 3 (1/2) (1/8)
3 % 4
*FDA> cut table 3 0 (1/2)
10923 % 32768
*FDA> cut table 7 0 (1/2)
19661 % 32768
*FDA> cut table 2 (1/2) (1/8)
5 % 8
*FDA> cut table 2 (1/2) (1/5)
22937 % 32768
*FDA> cut table 3 (1/2) (1/5)
29491 % 32768
```

```
*FDA> cut table 5 (1/2) (1/5)
20753 % 32768

*FDA> cut table 3 0 (1/2)

10923 % 32768
```

Type for “I cut, you choose”

In the implementation, the function for ”I cut, you choose” has the following type:

```
cutAndChoose :: Valuation -> Agent -> Agent
  -> Boundary -> [(Agent,Boundary,Boundary)]
```

The implementation assumes that the first argument represents the cutter and the second argument the chooser:

```
cutAndChoose f i j a = let
    b = cut f i a ((1-a)/2)
  in
    if eval f j a b >= eval f j b 1
      then [(j,a,b),(i,b,1)]
      else [(i,a,b),(j,b,1)]
```

Idea

Note that the idea is to fairly divide the part $[a..1]$ of the interval $[0..1]$. Half of this part should have value $\frac{1-a}{2}$.

Now we can illustrate our earlier point that if valuations are private, it is better to choose that to cut.

Agent 1 does the cutting

This is what happens if agent 1, who has uniform valuation, is the cutter:

```
*FDA> cutAndChoose table 1 3 0
[(3,0 % 1,1 % 2),(1,1 % 2,1 % 1)]
```

Agent 3 gets the part that is much more valuable to her:

```
*FDA> eval table 3 0 (1/2)
3 % 4
*FDA> eval table 3 (1/2) 1
1 % 4
```

Agent 3 does the cutting

If agent 3 does the cutting, this is what happens:

```
*FDA> cutAndChoose table 3 1 0
[(3,0 % 1,10923 % 32768),(1,10923 % 32768,1 % 1)]
```

Agent 3 gets a smaller part now. This is much better for agent 1, for:

```
*FDA> eval table 1 0 (10923/32768)
10923 % 32768
*FDA> eval table 1 (10923/32768) 1
21845 % 32768
```

Banach-Knaster, Implementation

To implement the Banach-Knaster algorithm, we need an auxiliary function for picking an element in a non-empty list with minimal f-value:

```
minim :: (Ord b) => (a -> b) -> [a] -> a
minim f = head .
          (sortBy (\ x y -> compare (f x) (f y)))
```

In the implementation of Banach-Knaster we use `minim` to pick the agent who cuts the smallest piece. The function `fromIntegral` converts `n` to a floating point number, which is needed for the division `(1/n)`.

```
bk :: Valuation -> [Agent] -> Boundary
   -> [(Agent,Boundary,Boundary)]
bk f [i] a = [(i,a,1)]
bk f js a = let
   n = fromIntegral (length js)
   g = \ j -> cut f j a ((1-a)/n)
   i = minim g js
   b = g i
 in
   (i,a,b) : bk f (js \\ [i]) b
```

An auxiliary function for evaluating all the pieces

```
evl :: Valuation -> [(Agent,Boundary,Boundary)]
    -> [(Agent,Value)]
evl table = let
    f = \ (i,a,b) -> (i, eval table i a b)
  in map f
```

Fairness

We call a cake division among N agents fair if each agent receives a slice that she values as at least $\frac{1}{N}$ of the cake.

```
fair :: Valuation -> Boundary
     -> [(Agent,Boundary,Boundary)] -> [Bool]
fair table a xs = let
    pairs = evl table xs
    n     = fromIntegral (length xs)
    f     =  \ (i,v) -> v + epsilon >= (a-1)/n
  in
    map f pairs
```

Illustration

```
*FDA> bk table [1,3] 0
[(3,0 % 1,10923 % 32768),(1,10923 % 32768,1 % 1)]
*FDA> fair table 0 $ bk table [1,3] 0
[True,True]
*FDA> bk table [1,3,5] 0
[(3,0 % 1,7281 % 32768),(1,7281 % 32768,40049 % 65536),
                        (5,40049 % 65536,1 % 1)]
*FDA> fair table 0 $ bk table [1,3,5] 0
[True,True,True]
```

Assertive Version of the Algorithm

We can use this to write an assertive version of the Banach-Knaster cake cutting algorithm:

```
bkA :: Valuation -> [Agent] -> Boundary
   -> [(Agent,Boundary,Boundary)]
bkA = let
  allFair = (\ f _ a outcome ->
              and (fair f a outcome))
  in assert3 allFair bk
```

Iterative formulation of the Banach-Knaster algorithm

Banach-Knaster algorithm for cake cutting (iterative version)

```
1. Let $N$ be the set of people sharing interval $[a..1]$.
2. While $|N| > 1$ do
    (a) let $i \in N$ be such that $b = \mathrm{Cut}(i, a, \frac{1-a}{|N|})$ is minimal;
    (b) give $[a..b]$ to $i$;
    (c) $N := N - \{i\}$;
    (d) $a := b$.
3. The remaining person gets $[a..1]$.
```

Implementation of the iterative version of the Banach-Knaster algorithm

```
bki ::  Valuation -> [Agent]
    -> [(Agent,Boundary,Boundary)]
bki f js = bki' f js 0 []

bki' ::  Valuation -> [Agent] -> Boundary
     -> [(Agent,Boundary,Boundary)]
     -> [(Agent,Boundary,Boundary)]
bki' f = while3
        (\ js _ _  -> not (null js))
        (\ js a xs -> let
            n = fromIntegral (length js)
            g = \ j -> cut f j a ((1-a)/n)
            i = minim g js
            b = if n > 1 then g i else 1
          in
            (js\\[i], b, (i,a,b):xs))
```

Another version, using a “for” loop

```
bki ::  Valuation -> [Agent]
    -> [(Agent,Boundary,Boundary)]
bki f js = let
    k = length js in
  bki' f k js 0 []

bki' ::  Valuation -> Int -> [Agent] -> Boundary
     -> [(Agent,Boundary,Boundary)]
     -> [(Agent,Boundary,Boundary)]
bki' f k = for3 [1..k]
        (\ m js a xs -> let
            n = fromIntegral ((k+1) - m)
            g = \ j -> cut f j a ((1-a)/n)
            i = minim g js
            b = if n > 1 then g i else 1
          in
            (js\\[i], b, (i,a,b):xs))
```

Analysis

Will a cake partition found by the Banach-Knaster algorithm always be fair? That depends on the properties of the valuations. For valuations that satisfy all requirements stated above, the answer is 'yes'. This can easily be proved by induction, as follows.

If there is only one agent, Banach-Knaster gives the cake fragment $[a..1]$ to that agent. This is fair by definition.

Suppose the Banach-Knaster division is fair for any cake fragment $[b..1]$ and n agents. We have to show that the Banach Knaster division is also fair for any cake fragment and $n+1$ agents.

Banach-Knaster, for $n+1$ agent set A and cake fragment $[a..1]$, instructs us to compute $[a..b]$, where $b = \min_{i \in A} \text{Cut}(i, a, \frac{1-a}{N+1})$, and give this to an $i \in A$ with $\text{Cut}(i, a, \frac{1-a}{n+1}) = b$.

Then $f_i[a..b] = \frac{1}{n+1}[a..1]$, so i gets a fair share of $[a..1]$. Now take some arbitrary $j \in A - \{i\}$.

Then $\mathrm{Cut}(j, a, \frac{1-a}{n+1}) \geq b$. Will we be able to give j a fair share of $[b..1]$? If the valuation function for j is proper, then the answer is yes, for then it follows from the fact that $\mathrm{Cut}(j, a, \frac{1-a}{n+1}) \geq b$ that $f_j[a..b] \leq \frac{1}{n+1}[a..1]$, and therefore,

$$f_j[b..1] = f_j[a..1] - f_j[a..b] \geq \frac{n}{n+1}[a..1].$$

By the induction hypothesis, j gets a fair share of $[b..1]$.

Some Background on Valuation and Measure

A valuation function with the properties mentioned on page 49 is often called a measure function.

Measure functions arise in probability theory as well, and it turns out that the valuation functions we need for expressing the individual appreciation of parts of a cake are related to so-called mass density functions in the same way as in probability theory.

A probability density function (p.d.f.) expresses the relative likelihood that a random variable takes a particular value. A probability density function is non-negative everywhere, and its integral is equal to one.

In our context, a mass density function is a function that is positive on any x in the interval $[0..1]$, and has an integral equal to one.

The uniform valuation (the valuation of agents 1 and 2 in `table`) arises from the mass density function $\lambda x \mapsto 1$. It is the function F that has $\lambda x \mapsto 1$ as its derivative, i.e., the function $F = \lambda x \mapsto x$. For if $F(x) = x$ then $F'(x) = 1$.

And so on.

Envy-Free Cake Cutting

We have seem that the cake divisions produced by Banach-Knaster are fair. Still, the results of the division may cause hard feelings among the participants. Suppose Alice gets her fair share which she considers worth $\frac{1}{3}$ of the cake. Next, Bob and Carol divide the rest. Both get a fair share: at least half of the rest, in their own evaluation. But Alice sees to her dismay that the piece that Bob considered a fair share is worth much more than the piece she got herself. (It follows that Alice also believes that the piece that Carol received is worth much less than her own piece, but such findings are generally easier to live with.)

Can we cut cakes in such a way that these feelings of envy are avoided? In the case of cake cutting for two, we can, for we know that "cut and choose" has as result that the cutter has one half of the cake (in his own estimation), while the chooser estimates to have at least one half. Neither of them wants to swap with the other.

In the case of three or more, matters are different. The example above shows that the Banach-Knaster algorithm does not guarantee envy-freeness, even in the simple case of sharing a cake with three participants.

Stromquist [14] describes the following algorithm for envy-free cake cutting for 3.

Envy-free Cake Cutting for Three: Moving Knives

“A referee moves a sword from left to right over the cake, hypothetically dividing it in a small left piece and a large right piece. Each player holds a knife over what he considers to be the midopoint of the right piece. As the referee moves his word, the players continually adjust their knives, aways keeping them parallel tot the sword [..]. When any player shouts “cut” the cake is cut by the sword and by whichever of the players’ knives happens to be in the middle of the three. The player who shouted “cut” receives the left piece. He must be satisfied, because he knew what all three pieces would be when he said the word. Then the player whose knife ended nearest to the sword, if he didn’t shout “cut,” takes the center piece; and the player whose knife was farthest from the sword, if he didn’t shout “cut,” takes the right piece. The player whose knife was used to cut the cake, if he hasn’t already taken the left piece, will be satisfied with whatever piece is left over. If ties must be broken — either because two or three players shout simultaneously or because two or three knives coincide — they may be broken arbitrarily.” [14]

Algorithm for envy free division for 3, attributed (in [4]) to Conway and Selfridge.

On next page.

Envy Free Division for 3 Players (Conway, Selfridge)

1. Call the measures of the three players μ_1, μ_2, μ_3.
2. Player 1 cuts cake into 3 equal pieces (according to μ_1).
3. If the two largest pieces according to μ_2 have unequal sizes, player 2 cuts the largest of them down to size by slicing off a piece L. This gives pieces X, Y, Z, and maybe a leftover piece L, trimmed from X.
4. Players 3, 2, 1, in that order, choose a piece. If 2 trimmed X, he has to choose X if player 3 does not choose it.
5. If there was no trimming we are done. Otherwise, let x be the player that received X, and let y be the other from players $2, 3$. Let y cut L into three equal pieces L_1, L_2, L_3 (according to μ_y).
6. The three pieces L_1, L_2, L_3 are divided by letting x choose first and y next, while 1 has to take the remaining piece.

Dividing an Estate, or Dividing a Burden

Suppose we want to divide m desirable objects (an ‘estate’) or undesirable objects (a burden) among n participants who are each entitled to an equal share of the estate (or obliged to an equal share in the burden). Each participant has her own valuation of the items. Can we make a division that is fair and envy-free? Each participant should receive at least $\frac{1}{n}$ of the estate, in her own estimation. Or in the case of a burden: each participant has to take care of at most $\frac{1}{n}$, in her own estimation. And (envy-freeness) no participant should be willing to trade her share with that of any other participant. Here is a simple example.

Alice, Bob and Carol have to divide an estate consisting of a cabrio car, a station car, a sailing boat, a grand piano and a collector’s wrist watch. Given that these items are so diverse, they each have different valuations for the objects. Is there a procedure for fair division that will not cause hard feelings?

It turns out that there is. The big equalizer is money.

Explain and implement the procedure proposed by Haake, Raith and Su in [8], where money is used to establish fairness and envy-freeness through internal market pricing.

Explanation by example

	Alice	Bob	Carol
car			
boat			
grand piano			

Splitting the Rent

My friend’s dilemma was a practical question that mathematics could answer, both elegantly and constructively. He and his housemates were moving to a house with rooms of various sizes and features, and were having trouble deciding who should get which room and for what part of the total rent. He asked, “Do you think there’s always a way to partition the rent so that each person prefers a different room?” [15]

If you want to stay informed . . .

The material of the course is undergoing further revision.

Send me an email: `jve@cwi.nl`.

References

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

[2] Steven Brams. Fair division. In Barry R. Weingast and Donald Wittman, editors, Oxford Handbook of Political Economy. Oxford University Press, 2005.

[3] Steven Brams. Mathematics and Democracy:Designing Better Voting and Fair Division Procedures. Princeton University Press, 2008.

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

[5] L.E. Dubisn and E.H. Spanier. How to cut a cake fairly. American Mathematical Monthly, 68(1):1–17, 1961.

[6] S. Even and A. Paz. A note on cake cutting. Discrete Applied Mathematics, 7:285–296, 1984.

[7] D. Gale and L. Shapley. College admissions and the stability of marriage. American Mathematical Monthly, 69:9–15, 1962.

[8] Claus-Jochen Haake, Matthias G. Raith, and Francis Edward Su. Bidding for envy-freeness: a procedural approach to n-player fair-division problems. Social Choice and Welfare, 19(4):723–749, 2002.

[9] Robert W. Irving. An efficient algorithm for the "stable roommates" problem. Journal of Algorithms, 6(4):577–595, 1985.

[10] Hervé Moulin. Fair Division and Collective Welfare. MIT Press, 2004.

[11] T.V. Padma. Mathematwist: Number Tales from Around the World. Tulika Publishers, Chennai, India, 2007.

[12] Jack Robertson and William Webb. Cake-Cutting Algorithms – Be Fair if You Can. A.K. Peters, 1998.

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

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

[15] Francis Edward Su. Rental harmony: Sperner's lemma in fair division. American Mathematical Monthly, 106(10):930–942, 1999.