The Flow Mapping Problem

Objectives

The primary objective is to partition *n* flows among *m* channels in a way that is:

- maxmin fair and
- wastes no usable channel capacity.

A secondary objective is to:

- balance channel loads

Problem Input

Input to the problem consists of three tables. The first defines the demands of each flow in arbitrary units. The seconds defines the capacity of each channel in the same units. The last is a two dimensional table in which each row defines that channels that the associated flow may access.

```
Flow demands
 22.00   8.00   5.00  17.00

Channel capacity
 10.00  10.00  10.00  10.00

Channel map
  1   1   0   0
  0   1   1   0
  0   0   1   1
  1   0   0   1
```

Problem Output

The mapping algorithm produces a two-dimensional table that maps portions of each flow's load to an available channel. Total allocation per flow is obtained by summing rows, and total load on each channel is obtained by summing columns.

	Channel				
Flow	0	1	2	3	
0	6.50	7.00	0.00	0.00	= 13.5
1	0.00	3.00	5.00	0.00	= 8.0
2	0.00	0.00	5.00	0.00	= 5.0
3	3.50	0.00	0.00	10.00	= 13.5
	10.00	10.00	10.00	10.00	

The solution is in general *not* unique

Relation to bonding groups

The channel map implicitly defines bonding groups.

- Two or more flows belong to the same bonding group i.f.f. the rows representing them in the map are identical.
- The number of bonding groups is the number of distinct rows in the map.

If there are multiple flows an a bonding group, the efficiency and effectiveness of the mapping algorithm is improved by consolidating the flows of each bonding group into a single mapping row whose associated demand is the aggregate demand of the flows in the bonding group.

The mapping algorithm will then compute the capacity allocated to the entire bonding group on each channel in the set. Since all flows in the bonding group have access to exactly the same channel set, it is straightforward to divide the capacity that is allocated by the mapping algorithm to the entire bonding group among the individual flows of the group in a maxmin fair (or other) way.

Combining the flows of a bonding group

In the problem below flows 1 and 2 belong to a single bonding group:

```
Flow demands
 14.00  15.00   5.00   6.00

Channel capacity
 10.00  10.00  10.00  10.00

Channel map
  1   1   0   0
  1   1   0   0
  0   0   1   1
  0   1   0   1
```

Rows are identical so flows 0 and 1 comprise a single bonding group

Thus the problem should be reformulated as:

```
Flow demands
 29.00   5.00   6.00

Channel capacity
 10.00  10.00  10.00  10.00

Channel map
  1   1   0   0
  0   0   1   1
  0   1   0   1
```

Load sharing groups

We say that the channel map represents a single *load sharing group* of flows and channels if and only if it is possible to travel from any point in the map that whose value is 1 to any other point in the map whose value is 1 by making either horizonal or vertical steps of arbitrary step-size through map points that also have the value 1.

This *map,* shown on the previous page, represents a single *load sharing group.*

```
Channel map
  1   1   0   0
  0   1   1   0
  0   0   1   1
  1   0   0   1
```

For example, to move from element [0][0] to element [2][2], a legal path is
`[0][0], [0][1], [1][1], [1][2], [2][2].`

To move from element [0][0] to element [3][3], a legal path is
`[0][0], [3][0], [0][3]`

Therefore, the algorithms presented here are applicable to channel maps consisting of a single *load sharing group* in which each row represents a *single bonding group.*

Multiple *load sharing groups* per map

A channel map, however, may define one *or more* load sharing group. If it does describe more than a single load sharing group, then the problem must be partitioned and each group processed individually.

For example the following map, describes three load sharing groups:

Group 0: Channels 0, 1 (flows 0, 1)
Group 1: Channel 2 (flow 3)
Group 2: Channel 3 (flow 2)

```
Channel map
  1   1   0   0
  0   1   0   0
  0   0   0   1
  0   0   1   0
```

By allowing flow 0 to use channel 3 as shown below it now contains only two load sharing groups:

Group 0: Channels 0, 1, 3 (flow 0, 1, 2)
Group 1: Channel 2 (flow 3)

```
Channel map
  1   1   0   1
  0   1   0   0
  0   0   0   1
  0   0   1   0
```

A group can also have more channels than flows
Group 0: Channels 0, 1, 2 (flow 0)
Group 1: Channel 3 (flow 1, 2, 3)

```
Channel map
  1   1   1   0
  0   0   0   1
  0   0   0   1
  0   0   0   1
```

Maxmin fair load assignment

For a single channel with multiple flows whose demands are specified in increasing order, maxmin fair allocation is a straightforward *O(n)* algorithm. Capacity is allocated to a single flow at each iteration of the allocation loop.

If the flow's demand is less than or equal to its fair share of the remaning capacity, it receives its demand. Otherwise it receives its fair share. At each iteration, the fair share is the remaining capacity divided by the remaining number of flows and after allocation the remaining capacity is reduced by the amount allocated.

```
/* If demands are ordered maxmin can be O(N) */

   left = CAPACITY;
   for (i = 0; i < COUNT; i++)
   {
      share = left / (COUNT - i);
      printf("%3d %6.0lf %6.0lf ",
              i, left, share);

      if (demands[i] < share)
         allocs[i] = demands[i];
      else
         allocs[i] = share;

      left = left - allocs[i];
      sum += allocs[i];
      printf("%6.0lf %6.0lf \n",
               demands[i], allocs[i]);

   }
```

For example assume capacity = 1000 and that the sorted demands are as follows:

```
double demands[] = {200, 260, 400, 500}
```

Then the output of the above program is:

```
  i   left   share demand alloc
---------------------------------
  0   1000    250    200    200
  1    800    267    260    260
  2    540    270    400    270
  3    270    270    500    270
```

The mapping algorithm

We will illustrate with a walkthrough of an example problem (*map3.txt*)

```
Original demands
 22.00   8.00   5.00  17.00

Channel capacity
 10.00  10.00  10.00  10.00

Channel map
  1   1   0   0
  0   1   1   0
  0   0   1   1
  1   0   0   1
```

Step 1: Compute adjusted demands.

If a flow demands more than can be delivered on all the channels that it has access to, the flow's demand is reduced to the sum of the capacities of the channels that it may use. Here flow 0 demands 22.0 but it has access to only 2 channels so its adjusted demand is 20.0.

```
Adjusted Demands
Flow   0      1      2      3
     20.00   8.00   5.00  17.00
```

Step 2: Compute upper bound on usable channel capacity.

This value is the minimum of the sum of the adjusted demands of all flows having access to a channel and the actual capacity of the channel. In this example, channel 2 is used by flows 1 and 2. Thus its maximum usable channel capacity is *min(8.0 + 5.0, 10.0).* The *aggregate channel capacity* is the sum of the individual maximum usable channel capacities. It is used to compute the maxmin fair target allocations to each flow.

```
Maximum Usable Channel Capacity
Chan   0      1      2      3
     10.00  10.00  10.00  10.00

Maximum Aggregate Channel Capacity
 40.00
```

Step 3: Compute maxmin fair allocation of aggregate channel capacity.

The *adjusted demands* are sorted and the aggregate capacity divided in a maxmin fair way.

```
Sorted Adjusted Demands
Flow  0     1    2    3
     5.00 8.00 17.00 20.00
```

Since there are 4 flows and 40.0 units of capacity, each flow's initial fair share is 10 units.

Thus flows 1 and 2 are allocated their demands of 8.0 and 5.0 leaving 27 units to be divided equally among flows 3 and 4.

```
Maxmin fair allocations
Flow  0      1      2      3
    13.50   8.00   5.00  13.50
```

Step 4: Compute maximum maxmin channel demand

For each channel this value is the sum of the maxmin fair allocations of each flow that can use the channel. Since flows 0 and 3 can use channel 0, its maxmin demand is 13.5 + 13.5

```
Maxmin channel demand
Chan   0      1      2      3
     27.00  21.50  13.00  18.50
```

I originally hoped that this approach would produce an optimal order for mapping flows onto channels. It seems to produce a reasonable ordering, but use of the ordering is not sufficient by itself to solve the problem.

Step 5. In increasing order of maxmin channel demand perform maxmin fair allocation of residual flow demand.

This step is the fairly grubby. The table *residual_demand[]* contains the unsatisfied portion of each flow's maxmin fair demand. As each channel is processed it is necessary to:

> Step5a: Count the number of flows that can use the channel and have positive residual demand
> Step5b: Process these flows in increasing order of *residual demand* performing maxmin fair allocation.
> Step5c: As capacity is allocated to a flow its residual demand is reduced.

```
Maxmin fair allocations
 13.50   8.00   5.00  13.50
Maxmin channel demand
 27.00  21.50  13.00  18.50
Channel map
  1   1   0   0
  0   1   1   0
  0   0   1   1
  1   0   0   1
```

Channel 2 is processed first. Its 10 units are evenly divided between flows 1 and 2. flow 1's residual demand is reduced to 3.0 and flow 2's is reduced to 0.

```
Final Allocation by Flow and Channel   Residual Demand
  6.50   7.00   0.00   0.00           13.50  8.00  5.00  13.50
  0.00   3.00   5.00   0.00
  0.00   0.00   5.00   0.00
  3.50   0.00   0.00  10.00
```

Channel 3 is processed next. Since flow 2 has no residual demand left, All 10 units are given to flow 3 and it is left with 3.5 units of residual demand.

```
Final Allocation by Flow and Channel   Residual Demand
  6.50   7.00   0.00   0.00           13.50  3.00  0.00  13.50
  0.00   3.00   5.00   0.00
  0.00   0.00   5.00   0.00
  3.50   0.00   0.00  10.00
```

Channel 1 is now processed. Flow 1 has only 3 units of residual demand so Flow 0 receives the other 7.

```
Final Allocation by Flow and Channel   Residual Demand
  6.50   7.00   0.00   0.00            13.50 3.00   0.00  3.50
  0.00   3.00   5.00   0.00
  0.00   0.00   5.00   0.00
  3.50   0.00   0.00  10.00
```

Finally, Channel 0 is processed. Flow 3 has only 3.5 residual units and the remaining 6.5 are assigned flow 0.

```
Final Allocation by Flow and Channel
  6.50   7.00   0.00   0.00              6.50 0.00  0.00  3.50
  0.00   3.00   5.00   0.00
  0.00   0.00   5.00   0.00
  3.50   0.00   0.00  10.00
```

Did it work?

To see if objectives were met, sum the final allocations across both rows and columns. Summing across each row yields the allocation by flow. These should match the maxmin fair allocations, and in this case they do

```
Final Allocation by Flow
13.50   8.00   5.00  13.50

Maxmin fair allocations
 13.50   8.00   5.00  13.50
```

Summing down each column yields load on each channel. These should equal the maximum usable channel capacity... And again they do.

```
Maximum Usable Channel Capacity
Chan   0      1      2      3
     10.00  10.00  10.00  10.00

Assigned Capacity by Channel
10.00  10.00  10.00  10.00
```

Not so fast

Although the algorithm often reaches the desired solution, in some cases it does not. In some cases (mapa.txt) it is simply impossible to attain global maxmin fair shares. The following map represents a single load sharing group. However, two flows that belong to a single bonding group have not been coalesced.

```
Original demands
 14.00  15.00   5.00   6.00

Channel capacity
 10.00  10.00  10.00  10.00

Channel map
  1   1   0   0
  1   1   0   0
  0   0   1   1
  0   1   0   1

Adjusted Demands
 14.00  15.00   5.00   6.00

Maximum Usable Channel Capacity
 10.00  10.00   5.00  10.00

Maximum Aggregate Channel Capacity
 35.00

Maxmin fair allocations
 12.00  12.00   5.00   6.00
```

Since high demand flows 0 and 1 share only channels 0 and 1 it is not possible to give them both their fair share of 12 units. However, the resulting assignments are "best case" fair.

```
Maxmin channel demand
 24.00  30.00   5.00  11.00
```

This example also illustrates the rationale for processing the channels in increasing demand order. If channel 1 were to have been processed before channel 3, then flow 3 would have been given 3.3 units on channel one producing a suboptimal assignment.

```
Final Allocation by Flow and Channel
  5.00   5.00   0.00   0.00
  5.00   5.00   0.00   0.00
  0.00   0.00   5.00   0.00
  0.00   0.00   0.00   6.00

Final Allocation by Flow
 10.00  10.00   5.00   6.00

Assigned Capacity by Channel
 10.00  10.00   5.00   6.00
```

It should also be noted that in this workload definition flows 0 and 1 comprise a single bonding group, but the problem was not properly reformulated. When it is reformulated the maxmin fair allocation to the bonding group is now 20.0 as it should be instead of 24.0 as before.

```
Original demands
 29.00   5.00   6.00

Channel capacity
 10.00  10.00  10.00  10.00

Channel map
  1   1   0   0
  0   0   1   1
  0   1   0   1

Adjusted Demands
 20.00   5.00   6.00

Maximum Usable Channel Capacity
 10.00  10.00   5.00  10.00

Maximum Aggregate Channel Capacity
 35.00

Maxmin fair allocations
 20.00   5.00   6.00

Maxmin channel demand
 20.00  26.00   5.00  11.00

Final Allocation by Flow and Channel
 10.00  10.00   0.00   0.00
  0.00   0.00   5.00   0.00
  0.00   0.00   0.00   6.00

Final Allocation by Flow
 20.00   5.00   6.00

Assigned Capacity by Channel
 10.00  10.00   5.00   6.00
```

Failure of the basic algorithm

Recall that the algorithm processed channels in order of increasing demand. This raises the question "what happens if maximum channel demands are equal?" It turns out that even when they are nearly equal we can show that the original algorithm doesn't always produce the desired solution.

```
Original demands
 24.00   8.00  12.00   7.00

Channel capacity
 10.00  10.00  10.00  10.00

Channel map
  1   1   0   0
  0   1   1   0
  0   0   1   1
  1   0   0   1

Adjusted Demands
 20.00   8.00  12.00   7.00

Maximum Usable Channel Capacity
 10.00  10.00  10.00  10.00

Maximum Aggregate Channel Capacity
 40.00

Maxmin fair allocations
 13.00   8.00  12.00   7.00
```

Maxmin demand on the channels are similar.

```
Maxmin channel demand            Residual Demand
 20.00  21.00  20.00  19.00    13.00   8.00  12.00   7.00
```

When the channels are processed in the order 3, 0, 2, 1. The following suboptimal allocation occurs. Channel 1 is not fully loaded and flow 2 does not receive its maxmin fair demand.

```
Final Allocation by Flow and Channel
  8.00   5.00   0.00   0.00
  0.00   3.00   5.00   0.00
  0.00   0.00   5.00   5.00
  2.00   0.00   0.00   5.00

Final Allocation by Flow
 13.00   8.00  10.00   7.00

Assigned Capacity by Channel
 10.00   8.00  10.00  10.00
```

Unlike the preceeding example in which the target allocation is infeasible, in this case the objectives can be attained by processing the channels in a different order. If the channels are processed in the order 3, 0, 1, 2.

```
Final Allocation by Flow and Channel
  8.00   5.00   0.00   0.00
  0.00   5.00   3.00   0.00
  0.00   0.00   7.00   5.00
  2.00   0.00   0.00   5.00
```

Processing the channels in the order 0, 3, 2, 1 also produces a correct but different solution.

```
Final Allocation by Flow and Channel
  5.00   8.00   0.00   0.00
  0.00   2.00   6.00   0.00
  0.00   0.00   4.00   8.00
  5.00   0.00   0.00   2.00
```

Possible solutions

- Identify an algorithmic approach for determining the proper channel processing order.
- Process all possible orders halting when a correct assignment is obtained.
- Develop an algorithm for redistibuting loads in a way that produces the optimal solution.

An algorithm for redistributing load could also be used for channel balancing, so that is where we focus.

```
Final Allocation by Flow and Channel
  8.00   5.00   0.00   0.00
  0.00   3.00   5.00   0.00
  0.00   0.00   5.00   5.00
  2.00   0.00   0.00   5.00
```

It the above suboptimal solution we see that flow 2 needs 2 additional units of capacity and channel 1 has an extra 2 units of capacity available. Thus we need to move 2 units of load from either channel 2 or channel 3 (the channels used by flow 2) to channel 1 and then allocate those units to flow 2.

There are two ways to do this in this example. The simplest is to move 2 units from flow 1 on channel 2 to flow 1 on channel 1.

```
Final Allocation by Flow and Channel
  8.00   5.00   0.00   0.00
  0.00   5.00   3.00   0.00
  0.00   0.00   7.00   5.00
  2.00   0.00   0.00   5.00
```

However, another way is to move 2 units from flow 3 on channel 3 to flow 3 on channel 0, then move two units from flow 0 on channel 0 to flow 0 on channel 1. This then allows us to add 2 units to flow 2 on channel 3.

```
Final Allocation by Flow and Channel
  6.00   7.00   0.00   0.00
  0.00   3.00   5.00   0.00
  0.00   0.00   5.00   7.00
  4.00   0.00   0.00   3.00
```

An algorithm for redistributing load

```
Final Allocation by Flow and Channel
  6.00   7.00   0.00   0.00
  0.00   3.00   5.00   0.00
  0.00   0.00   5.00   7.00
  4.00   0.00   0.00   3.00
```

We see that to move load from one channel to another we need to construct a path through the allocation table having the following characteristics

- It must start in a column from which load is to be removed and end in a column to which load is to be added.
- It must not touch the row representing the deficient flow.
- It consists of alternating horizontal and vertical steps.
- The first and last steps are always horizontal.
- Capacity is added to the target of a horizontal step and subtracted from the target of a vertical step.
- The above rule ensures that the total load on intermediate channels doesn't change
- Capacity changes must be constrained to prevent negative and/or excessive allocations

The rebalancing algorithm

The rebalancing algorithm is intended to be as general as possible in reassigning capacity. It hopefully can deal with multiple underallocated flows and channels but also doesn't get stuck in situations such as mapa.txt where there is no way to achieve balanced allocation. This is the purpose of the variable *moved.*

```
fid = find_low_flow();  // alloc < maxmin fair share
cid = find_low_chan();  // load  < max usable capacity
moved = 1
target = maxmin_alloc[fid] - flow_alloc[fid];

while (fid >=0 && cid >= 0 && moved)
{
   moved = 0;
   for each chan used by fid
   {
      for each flow not fid
      {
          success = make_path( from (chan, flow) to cid);
          if (success)
          {
               shifted = shift_load();  // the path is global
               final_alloc[fid][chan] += shifted
               moved = 1;
               if (shifted == target)
                  goto next_fid;
               else
                  target -= shifted;
          }
      }
   }
next_fid:
   fid = find_low_flow();  // alloc < maxmin fair share
   cid = find_low_chan();  // load  < max usable capacity
}
```

The *make_path()* function

As presently implemented the *make_path()* function is a simple recursive depth first search that constructs a path from the start point *(flow, chan)* to the target channel *cid*.

It uses the following global data structures in the obvious way.

```
typedef struct
{
   int fid;    // flow id
   int cid;    // channel id
}  pathel_t;

pathel_t path[MAXCHANS * MAXFLOWS];    // steps in the path
int      pathlen;                      // length of path
int      visited[MAXFLOWS][MAXCHANS];  // avoid cycles in path!
```

The *shift_load()* function

As previously described the path necessarily begins and ends with horizontal steps and consists of alternating horizontal and vertical steps between the first and last step.

The *shift_load()* function is passed the *target* amount of load to be moved from the channel at the start of the path to the channel at the end of the path.

It traverses the path *twice.*

During the first traversal the *target* is decremented as required to prevent a negative allocation.

During the second traversal it performs the actual alternating decrement and increment operations using the adjusted target.

This could be made more efficient by having the path construction algorithm compute the maximum amount of load to be shifted along the path.

Open Questions

Does this problem map directly to some well-studied problem in the graph algorithms domain?

---------- *if not* ----------

- Is there a better (or worse) definition of maxmin fair for this problem domain?
- Produce a technical definition of a correct solution which supports the mapa.txt sample
- Is the current solution (with rebalancing) provably correct?
- Is there a counterexample demonstrating that it is not correct?
- If not correct, does there exist a correct solution?
- What if any bound exists on the computational complexity of the correct solution?
- Does *step 5* contribute to the solution or will any simple heuristic work just as well ?