Applying Linear Programming Assignment Report

Benjamin Dunn

December 6, 2021

Contents

1 Introduction

Imagine a wall of bricks, these bricks can be arranged in any number of rows, and any number of bricks per row (length of the rows remain the same across the wall). Each brick has a positive or negative value and it is our job to maximize the total value of bricks in the wall. However bricks with bricks on top of them cannot be taken until the bricks above them have been taken.

For the assignment, we were required to use linear programming to help solve the above problem

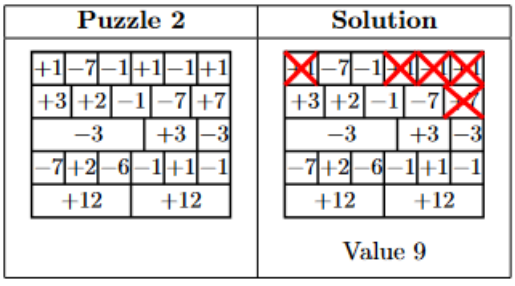

Figure 1: An example wall with solution.

and generate a solution for any possible wall of bricks. This report will cover how this problem can be modelled, how it was implemented into the given C program, and finally give an evaluation on if the assignment goals were achieved, what worked and what didn't work, and how it could be further improved given more time.

2 Modelling

2.1 Linear Program For Brick Problem

To begin, lets give an example problem then detail how we would set up a linear program to solve it.

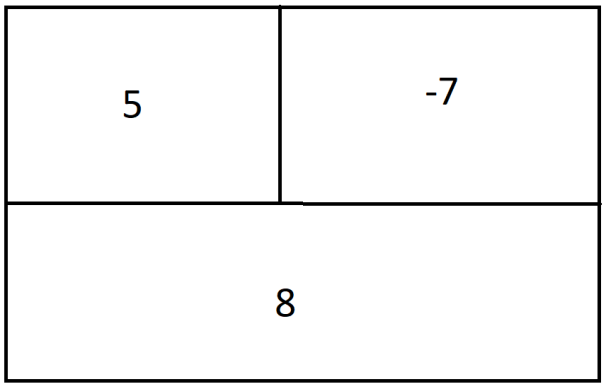

Figure 2: A simple example wall

In order to create a linear program for our problem we have to understand what our objectives are and what constraints exist within the problem. Our objectives here are to maximize total value of selected bricks, with constraints of not being able to pick bricks with bricks on top. This is a well known problem - the max closure problem. Further to this Jean-Claude Picard[1] described how to reduce max closure problems to max flow problems, a problem we know we can solve in polynomial time through linear programming. Using the methods described in Maximal Closure of a Graph and Applications to Combinatorial Problems[1] we can convert our closure problem to a max flow problem.

2.2 Max Closure to Max Flow

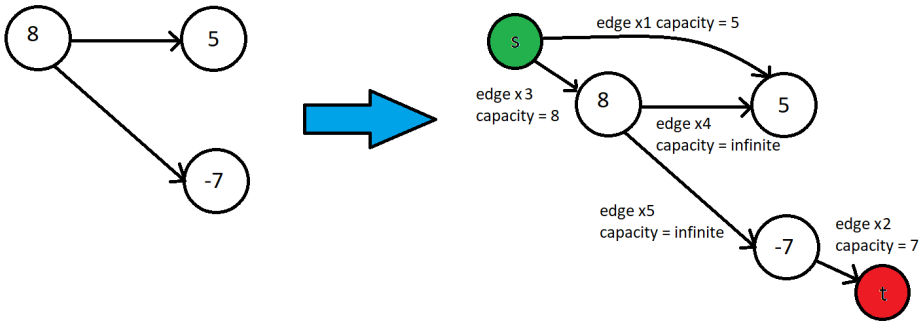

Figure 3: Figure 2's simple wall shown as a max closure problem, and then a max flow problem

Figure 3 clearly details the simple conversion between problems. Edges(the arrows) in max closure show the constraint of having bricks on top of other bricks (in this case the brick on bottom

with value 8 cannot be taken until bricks above, value 5 and -7, have been taken). Bricks are simply represented as vertexes(nodes etc). To convert we simply added a source and sink vertex and added edges to all positive and negative vertexes - edges from source to to all positive vertexes, edges to sink vertex from all negative vertexes. Edges from the source will have capacity of the value of the vertex/brick they are flowing to. Edges to the sink will have capacity of the value of the vertex/brick they are flowing from negated, e.g. -7 valued brick becomes 7 capacity edge flowing to sink. Edges between vertexes/bricks have infinite capacity, allowing flow to move freely through the brick problem. We now have a max flow problem with which to work with.

2.3 Max Flow as Linear Problem

Our linear problem must have an objective, variables, and constraints. The objective for the max flow is to maximize flow, variables refer to the capacities associated with edges and constraints being the direction of flow through each vertex. For max flow we can declare the objective in several ways:

- Maximize the sum of all flow along edges leaving the source. For our example this would be represented as: *Maximize x1 + x3*
- Maximize The sum of all flow along edges entering the sink For our example this would be represented as: *Maximize x2*
- Add an additional edge from sink to source and maximize the flow through it. In this case we would have to make sure to include the new edge in our constraints/variables.

For simplicity the objective can be written as Maximize: *x1 + x3*

Our variables can be written as:

$0 \leq x1 \leq 5$,

$0 \leq x2 \leq 7$,

$0 \leq x3 \leq 8$,

$0 \leq x4 \leq \infty$,

$0 \leq x5 \leq \infty$

All edges are lower bounded by 0 since we cannot have edges with negative flow.

Finally we must set up constraints. The number of constraints will be equal to the number of vertexes in the problem. To ensure all flow that enters a vertex leaves a vertex (where possible) we will set up constraints equal to zero e.g. Flow entering a vertex from edge1 must equal flow leaving the vertex on edge 2. For the example constraints would be:

x1 + x4 = 0,

x5 - x2 = 0,

x3 - x4 - x5 = 0

Using GLPK [2] we can set up these equations and very easily get the answer to the max flow problem. This isn't however the answer to the brick problem. To get that we must find the s-t cut which will provide the answer to our original max closure problem.

2.4 Max Closure/S-t cut

Once GLPK has ran its simplex function [2] to solve the max flow, we can retrieve the flow on our edges using function *glp_get_col_prim()*. When flow on our edge from source to vertex does not equal the total capacity of that edge we know this vertex is within the s closure and therefore within the max closure. We now know which positive value bricks are in the optimal solution and can therefore be selected.

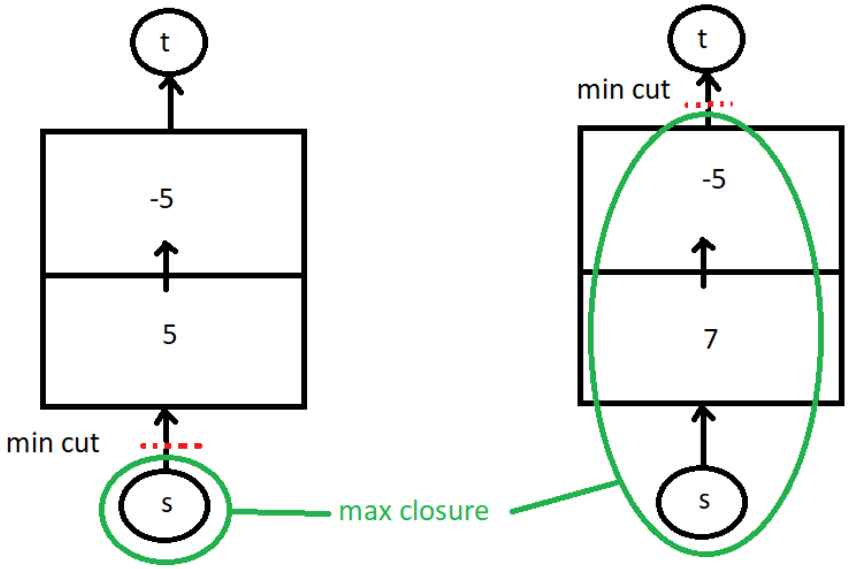

Figure 4: This example shows when flow on edges from source to vertex are not equal they are within the max closure. If flow is equal to capacity they are not in the closure since they are not worth taking.

Once all our positive bricks in the solution are selected simply ensure the rules of the problem are followed and select bricks above them. This will ensure all negative bricks that need to be taken are. In implementation make sure negative bricks that are chosen this way go on to select bricks above them as well. We now have an optimal solution to any brick wall problem.

3 Implementation

To implement the solution in the given C file the *computeSoluton()* function must update the given solution array and place 1's into the array where bricks are taken. To accomplish this I opted to make several loops to read detail how many bricks and edges there are in given puzzles.

Much of how these loops work are similar to how *printSolution()* reads through a text file, reading one brick at a time and adding the length of the brick to a variable tracking the current length of the row. Once the variable becomes greater then the max length of the row we know we are finished with the row and can move on to the next.

This looping is done again to count edges between bricks. Here however we need to track the length of bricks above/below the current row and by checking each loop if one of the three conditions are true:

- If lengths of the two rows are equal, add both lengths onto their respective length variables. Move the brick counters for their rows on by 1. Update the edge counter by 1.
- If length of the row above/below is greater then the current row increase the length of the current

row by its current brick. Move the brick counter for the current row by 1. Update the edge counter by 1.

- If length of the row above/below is less then the current row increase the length of the above/below row by its current brick. Move the brick counter for the above/below row by 1. Update the edge counter by 1.

When comparing the rows it could be set up to check either the above or below. In my solution it starts on row 1 as the current row and row 0 as the row above. Could have just as easily been row 0 as current row and row 1 as row below to compare to.

Now that we have the total number of edges in our problem (first loop counts edges from bricks to sources/sinks, second between bricks) we can set up the columns in the linear program (the variables) using *glp_add_cols()*. Then we simply loop through the edges again, setting them up as variables with *glp_set_cl_bnds()*. The bounds for these are from 0 to their brick value(absolute value to use the negative values in our problem). When we do this loop we make sure to add any positive value bricks to the objective function. This will represent the total flow passing from the source to the problem. As stated in the modelling section, this could also by the sum of all flow passing to the sink from the negative bricks, either would work.

One more loop through the problem will assign edge variables to the linear program that represent the edges from brick to brick. These edges have a lower bound of zero.

Now that the variables are set up we can deal with constraints. We know the number of constraints will be equal to the brick count (which we kept track of earlier when counting how many bricks we have) so we can use a simple for loop to start the constraints with *glp_set_row_bnds()*.

Actually setting up the constraints is a bit more involved. Using the loop that compares edge length and several variables to track which brick we are writing constraints for we can create a loop that will run through any problem and generate every constraint required.

```
edgeFromBrickToSourceOrSink = 1;
edgeFromBrickToBrickBelow = brickCount+1;
edgeFromBrickToBrickAbove = brickCount+1;

for (int i = 0; i < numRows; i++) {

    checkLenRowAbove = 0; // current length of upper row
    checkLenRowCurrent = 0; // current length of current row
    checkLenRowBelow = 0; // current length of lower row
    int j = 0; /* points to brick in upper row */
    int k = 0; /* points to brick in current row */
    int l = 0; /* points to brick in lower row */
```

Figure 5: Set up for the constraint loop. All rows except the first and last will need to tracks the length above and below the current row, hence the extra tracking variables.

The first thing that is checked inside the loop is whether the total number of rows is 1. If that is the case we can simply run through the single row and add the edge constraints for each brick, edge either directing flow from source or to the sink.

Now that the single row edge case is dealt with we can work on a normal problem with several rows. On the first row we need only check the row below it. So you link the brick to the source or sink

and then enter a while loop that will continue to add edges until the length of the bricks below the current brick are greater then it. At this point we know we can move onto the next brick and repeat the process. Using *glp_set_mat_row* at the end of the loop sets the constraint up. One important thing to note is the check:

```
//checks to see if brick lenghths are equal. If current row is not equal to lower row need to move row length
//back to ensure we are attatching all proper bricks to one another. Not going back means we may miss bricks
//if equal we know we will not miss any
if(checkLenRowCurrent != checkLenRowBelow){
    l--;
    checkLenRowBelow -= lengths[(i + 1) * length + l];
}
```

Figure 6: Important check to ensure brick to brick edges are not skipped.

As the comments describe, this ensures that when bricks of the row below have a greater total length then the current row we dont skip an edge. Moving back on the row we are comparing to ensures this. If the bricks in both rows make the current total length of the rows equal we can ignore the check. Otherwise we need to make sure the *checkLen* variable is corrected along with its brick counter.

This looping is repeated again for the final row in the problem, this time checking the last row to its row above, changing the checkLen, j,k,l, and respective checkLen variables to the relevant ones. This looping is done again for rows that are neither the first or last row. These check rows above and below for edges and update their respective constraints.

Now all variables and constraints are correctly generated from the given brick wall and can be solved with the *glp_simplex()* function. Now that the max flow problem is solved we can check where flow is moving through the problem. Bricks with positive value and where their corresponding edges from the source did not equal the total possible capacity are known to be within the max closure. We set these bricks with a 1 in our solution array. Negative bricks are chosen by picking any bricks above the known optimal solutions positive bricks. So any bricks that need to be selected to reach our optimal solution positive bricks are taken. This is done by simply looping through the brick wall, but starting on the last row first to ensure no bricks are missed that need to be taken. When a taken brick is encountered we also set bricks on top of the brick as taken as well. We now have a complete implementation for the assigned problem.

4 Evaluation

Overall I believe I produced a simple and complete solution to the task assigned. All ten text file examples can be solved and any additional custom ones. My modelling correctly demonstrates my knowledge and understanding of both max flow and max closure problems and how to move between the two. My implementation shows my understanding of how linear programs are set up with GLPK.

Whether or not using linear programming to help solve this problem is dependent on many other factors. What size of brick walls do we typically need to solve? If the brick walls are typically small then any computational time difference between a specialized algorithm and linear programming is likely negligible. Perhaps your time is limited and you don't have the time or resources to generate a faster specialized algorithm for the problem. Again in this case using linear programming is entirely reasonable and a very powerful tool.

With regards to my grade for this assignment:

- 10/10 for following the format guidelines. Documents guidelines were followed as instructed and on time.

- 8/10 for the Introduction section. I wrote a concise introduction to help those who are not familiar with the assignment understand the problem.
- 15/20 for the Modelling section. I believe I clearly explained how to model the problem. Perhaps more detail could have been added for explaining the s-t cut?
- 8/10 for the Implementation section. I believe I clearly explained how to implement the problem in the given C file. In conjunction with the comments in the C file it is clear how the solve function works.
- 10/10 for the Evaluation Section. I believe I clearly evaluated my work on the assignment, how effective linear programming was for the assignment and how well I think I did.
- 10/10 for the Reference section. Correctly referenced the GLPK documentation and the Picard 1976 paper as they were used to help with assignment.
- 10/10 for a C file that handles the known example files and correctly computes them. This was achieved.
- ?/10 for a C file that handles additional test files. It handled any of my custom walls, I have faith it can deal with yours as well!
- 6/10 for code readability. Could have made use of additional functions as loops were sometimes repeating code. However code is uniform in nature and well commented. Follows from each step naturally (building variables, building objective, building constraints, solving.)

So anywhere between a 77 or 87 I think would be fair.

On a side note I really enjoyed the assignment and only wish we had more classes to work with other advanced algorithm problems!

References

[1] Jean-Claude Picard. Maximal closure of a graph and applications to combinatorial problems. *Management Science*, 22(11):1268–1272, 1976.

[2] A. Makhorin. Glpk (gnu linear programming kit).
Available at `http://www.gnu.org/software/glpk/glpk.html`.