AMSI
VACATION
RESEARCH
SCHOLARSHIPS
2017-2018

Splitting Integer Linear Programs with a Lagrangian Axe

Robert Hickingobtham
Supervised by Andreas Ernst and Dhananjay Thiruvady
Monash University

Vacation Research Scholarships are funded jointly by the Department of Education and Training and the Australian Mathematical Sciences Institute.

Australian Government
Department of Education and Training

AMSI AUSTRALIAN MATHEMATICAL SCIENCES INSTITUTE

1 Introduction

Integer programming (IP) is an advance method to solve combinatorial optimization problems (COP) which are an important class of problems that have many applications in both industry and academia. A large number of these problems are NP-Hard or NP-Complete, as such, for large-scale COPs, it is infeasible to compute the optimal solution. Lagrangian Relaxation (LR) and Column Generation are methods to approximate the solution to the COP by solving an "easier" problem which gives a bound to the original IP problem. These methods requires a suitable set of constraints to be relaxed in order to formulate the relaxed problem. The set of constraints to relax are chosen manually which is typically a fairly time consuming process.

If the original constraints can be partitioned into "small", mutually exclusive sets B_i, where for any variable in a constraint in B_i, it is not a variable for a constraint in B_j where $j \neq i$, then each "block"; that is, a set of constraints and variable that no other block has overlap with; can be solved independently and in a much shorter time period. We therefore wish to determine a small set of constraints to relax so that the remaining constraints in the matrix can be partition into multiple blocks. This can be considered as a hypergraph problem where each vertex correspond to a constraint in the constraint matrix and each hyper-edge represents a variable. Alternatively, a Bipartite Graph $G = (Rows, Columns)$ can be used to visualise the problem where an edge (a, b) exists in G if and only if $A[a, b] \neq 0$. The problem then become determining a suitable "small" set of vertices S to delete from the hypergraph or graph G in order that $G - S$ contains a large amount of disconnected components with similar sizes.

In this paper, we developed different heuristic methods to automatically determine a set of constraints to relax and compare their computational time on different benchmark IP problems.

2 Literature Review

To the author's knowledge, no work has yet been done to develop methods to automatically determine a suitable set of constraints to relax for LR. However, methods for similar matrix decomposition problems have been developed for other areas of computational linear algebra. For example, **Nested dissection** [1] is a divide and conquer heuristic for solving systems of linear equation. This heuristic method works by using seperators to partition a graph G where the vertices in G represents the rows and columns of the matrix, and an edge in G represents a nonzero entry in the matrix. To determine the separators, the Graph G is planarise first then fragmented.

This idea of planarising a graph first then fragmenting is the main approach used in graph theory to fragment a graph [2]. In essence, the method involves removing vertices whose degree is high in some sense until the graph is planar. Then, use an approached based off Lipton & Tarjan's Planar separator theorem [3] to fragment the planar graph. Another method used to fragment graphs in a reasonable computation time is based on a Max-Flow Min-Cut approach on Multi-commodity Flow Problems in order to determine a suitable seperator. See [4] for a further explanation of this approach.

3 Base Algorithms

3.1 Decomposing the matrices into blocks

The following algorithm is used to decompose a constraint matrix into blocks given a set of constraints that are relaxed (equivalent to a set of rows that would be ignored).

Algorithm 1: Blocks

INPUT: $m \times n$ Constraint matrix A, and a set S of relaxed constraints

OUTPUT: Blocks - where each block B has a set of *rows* and *columns*

function BLOCKS(A,S)

1. **1. Intialise:**
2. $Rows = [1, 2, ..., m] \backslash \{S\}$
3. $Columns = [1, 2, ..., n]$
4. $Blocks := [\,]$
5.
6. **2. while** *Rows* is not empty **do**
7. Create a new Block B and any r from *Rows* to *B.rows*
8. Remove r from *Rows*
9. *new_cols* := [ ]
10. **for** c in *Columns* **do**
11. **if** A[r,c] $\neq 0$ **then**
12. Add c to *new_cols*
13. remove c from *Columns*
14. **end if**
15. **end for**
16. Add *new_cols* to *B.columns*
17. **while** *new_cols* is not empty **do**
18. *new_rows* := []
19. **for** c in *new_cols* **do**
20. **for** r in *Rows* **do**
21. **if** A[r,c] $\neq 0$ **then**
22. Add r to *new_rows*
23. remove r from *Rows*
24. **end if**

```
25.                 end for
26.             end for
27.             Add new_rows to B.rows
28.             new_cols := [ ]
29.             for c in Columns do
30.                 for r in new_rows do
31.                     if A[r,c] ≠ 0 then
32.                         Add c to new_cols
33.                         remove c from Columns
34.                     end if
35.                 end for
36.             end for
37.             Add new_cols to B.columns
38.         end while
39.         Add B to Blocks
40.     end while
```

Lines 6-40 is a While Loop that runs until each row has been allocated to a block. For each run in the cycle, the function works on a different block to allocate rows and constraints to.

Lines 10-14 is a For Loop that finds all the columns that has a non-zero entry in row r and adds them to the block B.

Lines 17-38 is a While Loop that runs until no new columns is added to the block **B**. Thus, a new block will be initialize if there are still rows that have not been allocated to a block yet.

Lines 19-26 finds any new rows that needs to be assign to the block B.

Lines 29-46 finds any new columns that needs to be assign to the block B.

3.2 Unrelaxing Constraints

This algorithm goes through the set of relax constraints S and determines a suitable set U to unrelax. A constraint $s \in S$ is unrelaxed if one of the following two criterias are satisfied:

1. The number of blocks does not decrease by relaxing s
2. Unrelaxing s does not result in multiple blocks merging to form a new block which has $> k$ constraints where k is specified by the user

Algorithm 1 Unrelaxing Constraints

INPUT: Set S of relaxed constraints; *Blocks* of the constraint matrix A; and k, the maximum size of a block

OUTPUT: Set U of constraints to unrelax

```
function UNRELAX(S, Blocks, k)
    U := []
    for s in S do
        SIZE := 1
        Overlap := []
        for B in Blocks do
            if A[s, :] ⊆ B.columns then
                Add s to U
                SIZE := k + 1
                BREAK
            else
                if A[s, :] ∩ B.columns is not empty then
                    SIZE := SIZE + size(B.rows)
                    Add B to Overlap
                end if
            end if
        end for
        if SIZE ≤ k then
            Add s to U
            Merge all blocks B in Overlap together
        end if
    end for
end function
```

4 Heuristic Methods for Determining Cut Sets

4.1 Random Relaxation

4.1.1 Basic Random Relaxation

The approach of this method is as follow: relax a random set of constraints; then, decompose the remaining, unrelaxed constraint into blocks.

Algorithm 2 Basic Random Relaxation

INPUT: $m \times n$ Constraint Matrix A

OUTPUT: Set S of constraints relaxed and *Blocks* of A

function BasicRandomRelax(A)

$Rows := [1, 2, ..., m]$

Choose a random set S from $Rows$

$Blocks := BLOCKS(A, S)$

end function

4.1.2 Advance Random Relaxation

The approach of this method is as follow: relax a "large" (10% of the total) set of constraints that are chosen at random; then, decompose the remaining, unrelaxed constraint into blocks. Go through and apply **Unrelaxing Constraints**. Add another large random set of constraint to the constraints relax then repeat the step above. Repeat

Algorithm 3 Advance Random Relaxation

INPUT: $m \times n$ Constraint Matrix A

OUTPUT: Set S of constraints to relax

function AdvRandomRelax(A)

$Rows := [1, 2, ..., m]$

$S := []$

while $Rows$ is not empty **do**

Choose a random set K of elements from $Rows$ of size $\sim \lceil m/10 \rceil$

Add K to S

$Rows := Rows \backslash \{K\}$

$Blocks := BLOCKS(A, S)$

$U := UNRELAX(S, Blocks, \sqrt{m})$

Remove U from S

end while

end function

4.2 Breadth-First Search Tree (BFS-Tree) approach

The algorithms that were developed using a BFS-Tree are based off the following two theorems.

Theorem 1: *Let* $\boldsymbol{T}$ *be a BFS-tree of a connected graph* $\boldsymbol{G}$ *with* $\boldsymbol{k}$ *levels. The set of vertices* J *on level* $\boldsymbol{j}$ *for all* $1 < \boldsymbol{j} < \boldsymbol{k}$ *is a cut-set of the graph* $\boldsymbol{G}$

Proof. Let v be a vertex on level $j+1$ of the BFS-Tree T rooted at vertex r. Since G is connected, there exist a $(r,v)-$path in G. As T is a BFS-tree, the shortest (r,v)-path has length $j+1$ since $dist_G(r,u) = dist_T(r,u)\,\forall\, u \in V(G)$.

Now consider the graph $G-J$. Suppose there exist a (r,v)-path in $G-J$. Consider the shortest path. This path will have length $l \geq j+1$. As it is the shortest path, for each $0 \leq d \leq l$, there exist a vertex $u \in (r,v) - path$ in $G-J$ such that the shortest (r,u)-path has length d. In particular, there exist a vertex u in the (r,v)-path that has a distance j from the root vertex r. But no vertices in the graph $G-J$ has distance j from the root vertex since they have all been deleted.

=> no (r,v)-paths exist in $G-J$

=> $G-J$ is disconnected

=> J is a cut-set □

Theorem 2: *Let* $\boldsymbol{T}$ *be a BFS-Tree of a connected graph* $\boldsymbol{G}$*. If a vertex* $\boldsymbol{v}$ *on level* $l>1$ *is a cut-vertex of* $\boldsymbol{G}$*, then there exist a vertex* u *on level* $j+1$ *such that*

1) $(v,u) \in E(G)$

2) $(j,u) \notin E(G)\ \forall\ j \in L\{v\}$

Where **L** is the set of vertices on level l

Proof. Let x and y be adjacent vertices to a cut-vertex v on G such that no (x,y)-path exist in $G-v$ (that is, they are on different components in $G-v$. If such vertices did not exist, then v is not a cut-vertex).Let T be a BFS-Tree rooted at r where v is on level $l > 1$ (where level 1 correspond to the root vertex).

As T is a BFS-Tree, it follows that vertices that are adjacent in G are either above, below, or on the same level as each other in T. Now suppose that neither x or y are on level $l+1$ (that mean that both x & y are either the same distance or closer to the root vertex then v).

=> there exist an (x,r)-path and a (y,r)-path in T that does not include v.

=> there exist an (x,y)-path in $T-v$.

=> there exist an (x,y)-path in $G-v$. This contradicts the assumption that no (x,y)-path exist in $G-v$.

=> either x or y is on level L+1.

Suppose just one of them is on level L+1. Assume, without loss of generality, that x is on level L+1. Suppose there exist a vertex z on level l that is adjacent to x in G.Therefore, there exist a (x,r)-path in G that does not include v. Additionally, since y is the same distance or closer to the root vertex then v, it follows that there exist an (y,r)-path in G that does not include v. Therefore, there exists a (x,y)-path in $G-v$, hence contradiction.

$\Rightarrow$ x satisfy the criteria for vertex u for *Theorem 2.*

Suppose both x and y are on level $l+1$. Assume that both of them are adjacent to vertices a and b respectively on level l that are not v (but, a may be equal to b).

$\Rightarrow$ there exist both an (a,r)-path and a (b,r)-path in G that does not include v.

$\Rightarrow$ there exist an (a,b)-path in $G-v$.This also is a contradiction.

It therefore follows that either x or y is not adjacent to any other vertex in level l. □

Both of these theorems were used as criterias to determine a suitable set of vertices to delete in order to fragment up a Graph. However, in practice, too many vertices were found that satisfy the condition of Theorem 1 making it ineffective to use to find a 'small' cut set. As such, for the algorithm that was developed, only vertices that satisfy the conditions of Theorem 2 were deleted.

Algorithm 4 BFS-Method

INPUT: $m \times n$ Constraint matrix A

OUTPUT: Set S of constraints to relax

function BFS_FRAGMENTING(A)

Construct a Bipartite graph $G = (Rows, Columns)$ where $(a,b) \in E(G)$ if and only if $A[a,b] \neq 0$

Intialise $S = []$

Choose a vertex v from $Columns$

Create a BFS-Tree T rooted at v

Find all vertices $u \in Rows$ in the BFS-tree that satisfy the condition of Theorem 2, add them to S

Blocks=BLOCKS(A,S)

$U := UNRELAX(S, Blocks, \sqrt{m})$

Remove U from S

end function

4.3 Deapth-First Search Tree (DFS-Tree) approach

To find the cut vertices within a connected hypergraph G, we utilise an algorithm based upon the following theorem of DFS-Tree.

Theorem 3: *For a DFS-Tree, a vertex u is a cut vertex if and only if one of the following two conditions hold:*
1) u is a root vertex of a DFS-Tree and it has at least 2 children
2) u is not a root of a DFS tree and and it has a child v, such that no vertex in the subtree rooted at v has a back edge to one of the ancestors (in the DFS tree) of u

Proof.

1. Let u be a root vertex of a DFS tree of a connected hypergraph G.

(=>) Suppose u has at least two children, v and w. Now consider the subtrees T_v and T_w of the DFS tree which are rooted at respectively v and w. Without loss of generality, assume v was picked before w in the DFS Tree algorithm. Suppose there is a hyperedge in G containing (x, y) where x $\in T_v$ and y $\in T_w$. If so, then the DFS tree algorithm would of picked this edge instead of choosing (u, w).

=> there is no hyperedge between any vertices in T_v with T_w.

=> there is no hyperedges containing (x, y) $\forall$ $x \in T_v$ and $y \in T_w \forall$ $v \neq w$ where v & w are children of u.

=> $u \in P \forall$ (v, w)-paths P in G. Therefore, there is no (v, w)-path P in $G - u$ so u is a cut-vertex

(<=) Suppose u has only one child. Therefore, it is a leaf of the DFS tree so $T - u$ is a spanning tree of $G - u$. Therefore, there is a x,y-path P $\forall$ $x, y \in V(G)$ and so u is not a cut vertex of G.

2. Let u be a vertex of a DFS-tree that is not the root.

(=>) Suppose u has a child v such that no vertices in the subtree rooted a v has a back edge to one of the ancestors (in the DFS tree) of u. Let T_v be the tree rooted at v

Suppose u has another child x in the DFS tree. Without loss of generality, assume that during the algorithm, x was chosen before v. Let T_x be the tree rooted at x. Now suppose in G, there exist an edge (a, b) where $a \in T_x$ and $b \in T_v$. However,if this was so, then the edge (a, b) would be picked instead of (u, v) but this didn't occur.

=> there exist no edge (a, b) with $a \in T_x$ and $b \in T_v$.

By this same line of reasoning it follows that no edge can exist between T_v and any other vertices in G besides the ancestor of u. Since no vertices in T_v has a back edge to the any of the ancestor of u, it follows that T_v is disconnected from the root vertex in $G - v$.

=> u is a cut-vertex.

(<=) Assume that for each subtree T_v of a child v of u has a back edge to one of the ancestor of u. Let r be the root of the T

=> for each vertex x in T_v for all child v of u, there exist a (r, x)-path in $G - u$.

=> for every y in $G - u$, there exist a (r, y)-path in $G - u$.

=> $G - u$ is connected. As such, u is not a cut-vertex

□

Algorithm 5 DFS-Method

INPUT: Constraint matrix A

OUTPUT: Set S of constraints to relax

function DFS_FRAGMENTING(A)

Construct a hypergraph G from A where each vertex in G correspond to a row in A and each hyperedge correspond to a column in A

Intialise $S = []$

Choose a vertex v

Create a DFS-Tree T rooted at v

Find all vertices u in the DFS-tree that satisfy the condition of Theorem 3, add them to S

end function

This algorithm is limited in that it can only detect cut-vertices within a graph G. However, most of the graphs that corresponded to the constraint matrices that were tested did not actually contain any cut-vertices. As such, this algorithm was found to be unworkable in practice since it was not able to determine any constraints to relax in order to decompose the constraint matrix into blocks.

5 Numerical Analysis of the Different Methods

5.1 Objective Variable

To compare the decompositions, we first define the *objective* variable as

$$Objective = \frac{|Blocks|}{sd(Blocks) * (\frac{|Relax|}{|Rows|})^2} \tag{1}$$

Where:

$|Blocks|$ = number of blocks

$sd(Blocks)$ = standard deviation of the block size

$|Relax|$ = Number of constraints relax

$|Rows|$ = Number of Rows

This variable gives an indication of how good a decomposition is. The larger the *objective* variable, the better a decomposition is.

5.2 Computational Results

Table 1 gives some basic statistic on the different constraint matrices which the decomposition methods were tested on

Table 1: IP Constraint Matrices

Instance	*#ofRows*	*#ofCols*	*nz%*
1	14	24	14.3
2	300	10000	0.67
3	233	2013	0.59
4	576	505	0.75
5	637	120	18.7
6	940	1480	0.21
7	2178	1156	0.42
8	1531	1680	2.79

$nz\%$ = Percentage of non-zero elements in the sparse matrix

Table 2 gives data about the decompositions that were obtained for each constraint matrix using the Advance Random Relax Method.

Table 2: Advance Random Relaxm Method (Best out of 10)

Instance	$\lvert Blocks\rvert$	$\lvert RowsRelax\rvert$	$sd(Blocks)$	*Objective*	*CT*
1	12	2	0	588	0.013
2	92	191	1.77	128	6.9
3	93	36	3.59	1085	1.32
4	1	–	–	–	0.62
5	1	–	–	–	1.83
6	140	284	9.3	165	3.25
7	1	–	–	–	3.84
8	3	3	861	907	13.1

Table 3 gives data about the decompositions that were obtained for each constraint matrix using the BFS-Method.

Table 3: BFS-Method

Instance	$\lvert Blocks\rvert$	$\lvert RowsRelax\rvert$	$sd(Blocks)$	*Objective*	*CT*
1	12	2	0	588	0.0017
2	92	191	1.77	128	0.33
3	79	71	3.8	224	0.033
4	54	214	8.56	46	0.038
5	1	–	–	–	0.426
6	125	248	10.14	177	0.0812
7	2	1992	49.5	0.048	0.59
8	126	117	92.3	234	2.43

CT = Computational Time (s)

5.3 Comments on the computational results

Both the *Advanced Random Method* (ARM) and the *BFS-Method* were able to obtain decent decompositions for most of the *instances* that were tested. For *instance 5*, neither methods were able to obtain a decomposition, however, this was not unexpected since *instance 5* has a high percentage of non-zero elements (18.7%). As such, the corresponding graph to the constraint matrix is likely to be too connected in order to easily determine a small set of vertices to delete to fragment it up.

In contrasting these two methods, BFS-Method seems to be the best method of the two as it was able to obtain a decomposition for more *instances* then the ARM method were able to (3 times ARM was not able to obtain a decomposition as oppose to just once for BFS-Method). However, the decompositions that ARM was able to obtain tended to be either better or equal to that obtain by the BFS-Method (ARM had a much higher *objective* value for instance 3 and 8).

6 Conclusion and Further Work

Both the advance Random Relaxation Method (best out of 10) and the BFS-Method were able to generate reasonable decompositions of the constraint matrices. Work is currently being done on implementing these decomposition methods to determine how impactful they really are to the overall computation time of LR.

If methods of automating the decomposition process do seem to have potential in improving LR, further work can be done by developing more advance methods based upon more efficient, graph fragmenting algorithms (see [2] and [4] for examples).

7 References

1. George, A., 1973. Nested dissection of a regular finite element mesh. SIAM Journal on Numerical Analysis, 10(2), pp.345-363.

2. K J Edwards and G E Farr, Graph fragmentability, in: L Beineke and R Wilson (eds.), Topics in Structural Graph Theory, Cambridge University Press, 2013, pp. 203–218.

3. Lipton, R.J. and Tarjan, R.E., 1979. A separator theorem for planar graphs. SIAM Journal on Applied Mathematics, 36(2), pp.177-189.

4. Leighton, T. and Rao, S., 1988, October. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In Foundations of Computer Science, 1988., 29th Annual Symposium on (pp. 422-431). IEEE.