Randomized Rounding: A Primal Heuristic for General Mixed Integer Programming Problems

Mahdi Namazifar*, Robin Lougee-Heimer **, Andrew Miller ***, and John Forrest †

August 2009

Abstract. We propose an algorithm for generating feasible solutions to general mixed-integer programming problems. Computational results demonstrating the effectiveness of the heuristic are given. The source code is available as a heuristic in the COIN-OR Branch Cut Solver at www.coin-or.org

Key words: Randomized rounding, primal heuristics, mixed integer program, random walk.

1 Introduction

General mixed-integer progamming problems are an important class of optimization problems due to their wide-spread application. Business applications that can be successfully modeled as general mixed-integer programs include manufacturing production planning, capacity planning problems in telecommunications, (more), and portfolio optimization. As each generation of optimization tools is deployed, its success is met with an ever increasing demand to solve larger and more complex problems. Due to the softness of constraints and quality issues with input data, it is often not necessary (or even desireable) to solve problems to optimality. A good feasible solution found quickly can be useful as a final answer. Moreover, if an optimital solution is desired, a good initial feasible solution can provide a bound that can be used to fathom nodes in a search tree. For these reasons, among others, many researchers have focused their attention on heuristics to construct feasible solutions quickly. In this paper we propose a new algorithm for finding primal feasible solutions to the following general mixed-integer programming problem.

* Corresponding Author. Department of Industrial and Systems Engineering, University of Wisconsin-Madison. Email: namazifar@wisc.edu

** Business Analystics and Mathematical Sciences, I BM T. J. Watson Research Center. Email: robinlh@us.ibm.com

*** Institut de Mathematiquest de Bordeaux, Unversite Bordeaux. Email: Andrew.Miller@math.u-bordeaux1.fr

† Business Analystics and Mathematical Sciences, IBM T. J. Watson Research Center. Email: jjforre@us.ibm.com

Others

$$(MIP) : max\{cx + hy : Ax + Gy \leq b, x \in Z_+^n, y \in R_+^p\} \tag{1}$$

The set $S = \{x \in Z_+^n, y \in R_+^p, Ax + Gy \leq b\}$ is the feasible region. A point $(x, y) \in S$ is a feasible solution. The set $\bar{S} = \{x \in R_+^n, y \in R_+^p, Ax + Gy \leq b\}$ is the continuous relaxation of S, and the linear programming (LP) relaxation of MIP is

$$(LPR) : max\{cx + hy : Ax + Gy \leq b, x \in R_+^n, y \in R_+^p\} \tag{2}$$

1.1 Primal heuristics

Heuristics can be constructive or improvement. An improvement heuristic takes a(n integer) feasible point as an input and seeks to improve it. A constructive heuristic produces a(n integer) feasible from scratch. Our focus is on constructive hueristics.

Examples of constructive primal heuristics:

- Rounding. Solve LPR. Let $(\bar{x}, \bar{y})$ be the optimal solution to LPR. For every i such that $\bar{x}_i \in Z$, fix x_i based on a rounding of $\bar{x}_i$. Solve the reduced MIP. Rounding is one of the most basic heuristic operations. Rounding heuristics are widely employed for specialized problems, for example the simple greedy heuristic for the knapsack problem (ref. Nemhauser and Wolsey, pg 440) Rounding is also used successfully to construct valid inequalities to strengthen the linear programming formulationsChvatal-Gomory cuts (ref. Nemhauser and Wolsey).

- Diving.

- LP-and-Fix. Solve LPR. Let $(\bar{x}, \bar{y})$ be the optimal solution to LPR. For every i such that $\bar{x}_i \in Z$, fix $x_i = \bar{x}_i$. Solve the reduced MIP

- Relax-and-Fix

- Feasibility Pump [14]. Given a point $(\bar{x}, \bar{y}) \in \bar{S}$, round $\bar{x}$ to obtain a new point, $(\dot{x}, \bar{y})$. Repeat the following steps: (i) Find the closest point in $\bar{S}$ to $(\dot{x}, \bar{y})$, say $\tilde{x}, \tilde{y}$. (ii) Round $\tilde{x}$. (Note: there was the original FP by Bertacco et al (2007) and the objective FP by Achterberg and Berthold (2007) both availalble on NEOS.).

- Analytic Center Feasibility Method [10]. The ACFM has three phases:
  1. The weighted analytic center is found and a search for feasible soultions is conducted.
     - Use Newton's method to find the analytical center.
     - Find the extreme points that max and min the LP relaxation

- On the line segment from the analytic center to the min-extreme point (and likewise on the line from the analytic center to the max-extreme point): (i) take a step, (ii) round all the integer componets, (iii) fix the integer variables to their rounded values, (iv) solve the resulting lp. If the reduced problem is feasible, then its optimal solution is feasible to the original problem. Note: Bounds can be tightened and the algorithm re-run to find better feasible solututions.

2. If (1) does not produce s feasible integer solution, a second phase is started where the weights of the violated constraints are changed to to shift the analytic center (hopefully into the localization set).
3. If neither (1) or (2) produces an feasibe integer solution, a minimum distance problem is solved

Examples of improvement primal heuristics include Relaxation Induced Neighborhood Search (RINS) [3], Local Branching [4] and Solution Crossing [15].

1.2 The structure of the paper

The paper is organized as follows. random walk in a convex body, description our our randomized rounding heuristic, computational results, and summary.

2 Random Walk in a Convex Body

Randomly sampling from a convex body is a well-studied problem which has many different applications. Generally the goal is to obtain a random walk which gets close to its stationary distribution in a polynomial number of steps. Typically, uniformity of the distribution or other nice property is desired. One application using random walks is the problem of computing the volume of an n-dimensional convex body. In [12], similiar to a Monte Carlo Method, the solution approach is to find random points inside the convex body which are uniformly distributed. Another application such random sampling is finding the permanent of a matrix with non-negative elements. In [13], the solution approach taken is to sample from a combinatorial structure. The combinatorial structure used is not convex, but the concept employed is closley-related to that of sampling from a convex body. Portfoilo management is another application where researchers have applied techniques using random walk in a convex body [11]. Recently, Bertsimas and Vempala [2] proposed an algorithm for solving convex programs using random walks.

In the literature, many papers concerning random walk in a convex body assume the convex body is given by an oracle. In the context of mixed-integer (linear) programming, the feasible region of the linear programming relaxation is the convex body. The oracle in this conext is completely specificed by $\bar{S}$.

We briefly describe the most frequently used random walk algorithms.

2.1 Review of existing algorithms for random walk in a convex body

Ball-Steps- In this method, we choose a step size and a random initial point inside the convex body. At each iteration, a vector from the unit ball of the dimension of the convex body is selected and we move from the current point towards the randomly selected vector by the step size. The membership of the resulting point to the convex body is checked by calling the oracle. If this point is in the convex body, it becomes the current point and we iterate. Otherwise, we pick another random vector from the unit ball and follow the above procedure.

As you can see, in this method at each iteration the oracle is called once. Suppose that we want to generate random points in the polyhedron of a mixed-integer program. We would need to have a matrix-vector multiplication to show that a point is inside the polyhedron. This means that to generate each random point we would need $O(n^2)$ operations. This fact makes it computationally expensive to use the Ball-Steps method to generate random points inside the polyhedron of a mixed integer program.

Hit and Run- In this method, we pick a random initial point inside the convex body. At each iteration we pick a random vector from the unit ball and from the current point we go towards the direction of this vector and its negative until we hit the boundary of the convex body. Because of convexity properties, we have a line segment inside the convex body. We randomly select a point on this line segment, which becomes the starting random point in the next iteration of the algorithm.

If we are dealing with a polyhedron which is defined by half-space (the case in mixed-integer linear programm) to find the points where the line from the current point towards the vector intersects with boundary of the convex body, we need to a matrix-vector multiplication at each iteration (that means $O(n^2)$ operations for each random point) which is computationally expensive.

Walking on a truncated grid- In this method we define a grid which is sufficiently fine and an initial random point. At each iteration from the current point one goes one unit of the grid towards one of the possible directions (two times the dimension of the convex body different directions) and makes sure that by moving towards that direction, the point stays in the convex body.

The methods described above are known to have polynomial convergence to the stationary distribution. For the purpose of this paper, we do not need uniformity of the generated points. Instead, we seek a way to find random points inside the convex body very quickly. To accomplish this purpose, we present a method of generating random points inside a polytope with known extreme points, and later on we will discuss why we are interested in finding random points inside such a polytope.

2.2 Our random walk algorithm

For a purpose that will be clear shortly, suppose that all we know from convex polytope is its extreme points. Based on Minkowski's theorem, we know that the extreme points of a polytope are sufficient to describe that polytope. We generate random points inside this polytope by the following algorithm.

1: Construct an initial point in the interior of the polytope. One generation method to construct the initial point is to take a random convex combination of the extreme points. Let this point be the current point.
2: **while** (Number of points is less than desired) **do**
3: Randomly select a extreme point.
4: On the line connecting the current point and the randomly selected extreme point, randomly pick a point and take it as the current point.
5: **end while**

This algorithm is computationally efficient. Each iteration only requires a vector summation and a constant-vector multiplication which means $O(n)$ operation per random poin. Using this algorithm, we can generate numerous random points inside the polytope very quickly.

3 Randomized Rounding

In this section we explain how our proposed heuristic, Ranomized Rounding, works. From a high level prespective, Randomized Rounding first tries to find some of the extreme points of the feasible region of the problem. Then it tries to find random points in the interior of the polytope for which we have all the extreme points (the points we found in the first step). By rounding these random points to the nearest norm-1 integer point we hope to find integer feasible solutions for the problem. In the remaining of this section we explain how we do the above mentioned in more details.

As we described earlier, we need to find some of the extreme points of the feasible region of the problem. To do so we can use the primal simplex method by which at each iteration we get to a new extreme point (if it is not a degenerate pivot). So we keep the solution of each iteration of the primal simplex method. For the starter we solve the LP relaxation of the problem by primal simplex and we get a set of extreme points of the feasible region. But we still need a lot more extreme point which are some what diverse in the sense that they give us a rather uniform sample of the set of all the extreme points of the feasible region. By this we mean that we want a set of extreme points which are from all over the feasible region. To get such a set of extreme points, we build some random vectors which are generated by randomly tilting the constraints of the problem. Then we replace the objective function of the problem with these random vectors and we set the sense of objective based on the sense of the constraint we tilted (maximization for less than or equality constraints, minimization for greater than or equality constraints, and one of the two for equality constraints). Then we start to solve this new problem (which basically has just a different objective

function compared to the original LP relaxation) using primal simplex again, and at each iteration we keep the solution. We do this until we reach the maximum number of extreme points parameter or until we have no more constraints to tilt.

An important implementation point raises here; we need to pick the constraints in a random order. The reason for this is usually in the MIP models, consecutive constraints are pretty similar or related to each other. This fact causes the consecutive LP problems to have the same (or very close) optimal solutions, and hence if we start solving each new LP using the previous LP's optimal basis we won't have many different extreme points at each LP solve. On the other hand, if we shuffle the constraints this problem goes away.

One other important implementation issue is how to randomly tilt a hyperplane defining a constraint. In Randomized Rounding for a given constraint we look at the coefficients of the hyperplane defining it. If the coefficient is 0 we randomly make it 0.1 or -0.1. If the coefficient is not 0 randomly we multiply it by either 1.1 or 0.9. This gives us a tilted hyperplane which can be used to set up a new objective funtion. The reason why we tilt the constraints in the first place is first of all we try to make the LP problems to have a single optimal solution; and second of all we want to avoid the case in which one point is optimal to a buch of LP problems which makes the process of getting the extreme points to visit the same point over and over again.

At this point we have a set of extreme points of the feasible region of the LP relaxation of the MIP problem. The first thing we do next is check the feasiblity of these points to the MIP. If any of these points ends up being feasible, we check if it's better than the best solution we have found yet, and if so we update the best feasible solution.

After this we start generating random points inside the convex hull of these extreme points using the method which was described in the previous section. After generating each random point we round it to the norm-1 nearest integer point and check its feasibility. If it's not feasible we generate the next random point and so on. If it's feasible we check whether it's better than the best feasible solution found yet and if so, we update the best feasible solution. We generate random points until the number of generated random points exceeds a parameter.

One other important technical point is when we are solving one of the linear programs generated by tilting one of the constraints, the primal simplex dos not have to go all the way to find the optimal solution. For some problems it takes too many iterations for primal simplex to solve the linear programs. For such cases we can abort the primal simplex algorithm after a number of iterations and start working on the next linear program. This will help the heuristic to find more diverse extreme points which are found using a more diverse set of objective functions.

The high-level outline of our Randomized Rounding heuristic is as follows.

$S \leftarrow 0$

– Construct a subset of extreme points of $\bar{S}$

- Select a row of the constraint matrix
- Create an objective function based on the row
- Invoke the primal simplex method to solve the LP relaxation of the orginal problem with the new objective. Abort the LP solve after a certain number of primal simplex iterations.
- At each iteration of the LP solve, store the extreme point found.
- Do this until the desired number of extreme points have been found or until every row has been used.

– Construct random points inside the convex hull of the set of extreme points we have found.

- Initialize the point to be a random convex combination of the set of extreme points.
- Repeat the following until the desired number of random points is reached.
- (i) Choose a random extreme point from the set of extreme points
- (ii) Set the new point to be a random point on the line segment connecting the current point and the random extreme point.
- (iii) Round each random points to the closest norm-1 integer points
- (iv) Evaluate the original objective function at the rounded point. If it is attractive, check evaluate the points feasibility. (This eliminates the unnecessary computation expense of evaluting feasibilty for points that are unnattractive.)
- (v) Checking the feasibility of these integer points
- Do this until the desired number of random points is reached.

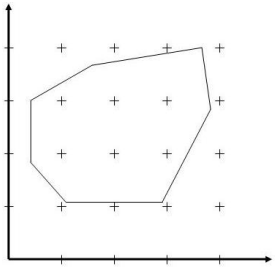

Fig. 1: The feasible region of the problem. Both variables are integer.

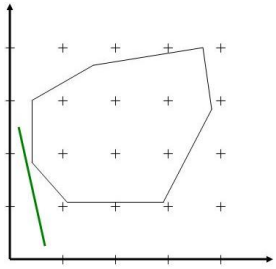

Fig. 2: Generate a random objective function by tilting one of the constraints

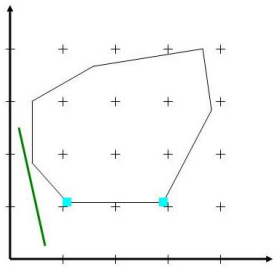

Fig. 3: Solve the LP using primal simplex. We don't necessarily have to solve to optimality.

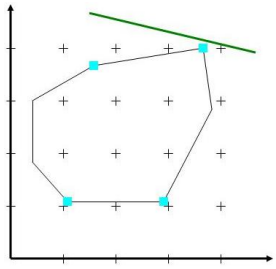

Fig. 4: Generate another random objective function and repeat the process untill you have enough points.

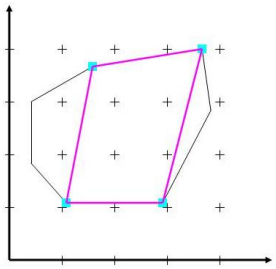

Fig. 5: Take the polytope defined by the extreme points you have found.

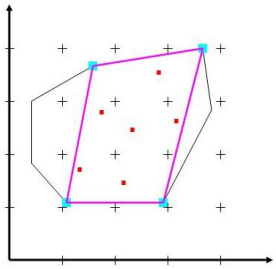

Fig. 6: Using the method presented, find random points inside the polytope.

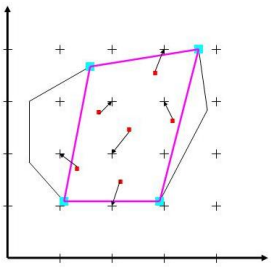

Fig. 7: Round the random points you have found to the closest integer feasible point.

As one might predict, this heuristic might not work well in the cases where the problem is far from full dimensional. In such problems when we round the randomly generated points with a very high chance the rounded points fall out of the feasible region of the problem and hence become infeasible. In fact for problems that are not close to full dimensionality it is very unlikely to find an integer feasible solution using this technique.

The main reason of losing dimensionality for linear programs is having equality constraints in the formulation of the problem, and also havin inequality constraints that are always satisfied at equality. This could give us a touchstone to check if Randomized Rounding is suited for a specific problem. In other words, by looking at the number of equality constraints plus the number of equlity constraints that are always satisfied at equality, and compairing it with the total number of variables we can have an idea of whether Randomized Rounding would be successful on the problem or not. As we will see in computational re-

sults section for close to full dimensional problems this heuristic can performs well and find good feasible solutions fast.

It is trivial to find the number of equlity constraints. On the other hand, finding the number of inequality constraints that are always satisfied at equality is a bit tricky. Consider one inequality costraint ($<=$) and suppose that we want to determine whether it is always satisfied at equality. To do so, we can form a linear program which is the same as our original problem except it has a slack variable for the constraint of our discussion and its objective function is to maximize this slack variable. If the optimal value to this linear program is zero, we can conclude that this inequality is always satisfied at equality. By doing this for all the inequality constraints one by one, we can find out how many of the inequality constraints are always satisfied at equality.

In the method mentioned above we might end up solving many linear programs to figure out how many inequality constraints are satisfied at equality. In [5] it is shown that by just solving one linear program one can find this number. To do so, consider the original linear program and add a slack variable for each inequality constraint and impose that these slack variables must be smaller than one. Also change the objective function to maximize the sum of the slack variables. Finally to ensure that any of the slack variables that can be nonzero can reach their upper bound of 1 we add another variable which is multiplied to the right hand side of the inequality constraints. Here is the formulation of this auxilary linear program:

$$
\begin{aligned}
(AuxilaryLP) \qquad & \max \textstyle\sum \delta_i \\
& \text{s.t. } Ax + I\delta \leq b\lambda \\
& \lambda \geq 1 \\
& 0 \leq \delta_i \leq 1,
\end{aligned}
$$

considering that the original problem is the following:

$$
\begin{aligned}
(OriginalLP) \qquad & \max c^T x \\
& \text{s.t. } Ax \leq b.
\end{aligned}
$$

3.1 Modified Randomized Rounding

As we noted earlier, Randomized Rounding works for the problems which are close to full dimensionality. In the rest of this section we present a modification of Randomized Rounding which is able to find feasible solutions for problems which are not close to full dimensionality. This modified heuristic is very similar to the original Randomized Rounding. The main difference is that in Randomized Rounding, after rounding the randomly generated points we check their feasibility. In the modified version of the heuristic, after rounding each randomly generated point we form a linear program. This linear program is in fact the LP relaxation of the original problem with the integer variables fixed to the values they take at the rounded point. In another word, for each rounded point we try

to find a point in the feasible region of the LP relaxation problem with the property that its integer dimensions take the same value as the integer dimensions of the rounded point.

Therfore, In this modified heuristic for each rounded point we solve a linear program. If the linear program is infeasible we can conclude that there is no integer feasible solution to the problem with the integer variables taking the same values as the rounded point. On the other hand, if the linear program is feasible, solving the LP gives us the feasible point with the same integer values as the rounded point which takes the best objective value. It is clear that by doing this, we are more likely to find feasible points, due to a more systematic approach towards finding integer feasible solutions based on randomly generated points inside the feasible region of the LP relaxation of the problem.

The drawback of this heuristic is we need to solve many linear programs. In this regard, on the other hand, we need to remember that most of these linear program solves are technically just checking the feasiblity of the program; whcih perhaps is computationally not as expensive as solving the LP relaxation of the problem. But still we are solving (checking feasibility of) such linear programs for each random point that we generate. Therefore, cosidering the number of the random points we generate, we might end up expending so much computation on solvin these linear programs. This fact suggests that if the LP relaxation of the problem is large, using this heuristic might not be computationally efficient. We discuss this in more details in the next section.

4 Computational Results

4.1 Implementation and Test Beds

Randomized Rounding and its modified version are implemented in COIN-Cbc [8]. There are a few implementation issues that we will discuss here. First of all, a valid question could be at most how many extreme points should we find before starting to find randomly generated points. The fact is we don't want to end up spending too much time finding the extreme points. On the other hand, we would need a diverse and big enough set of extreme points. This maximum number of extreme points we need to find of course depends on the size of the problem and many more parameters. In our experiments we didn't explore for such relations. Instead we just used 10000 as the maximum number of extreme points we find before we go to the next step.

As was mentioned in the previous section, Randomized Rounding does not perform well on problems that are not close to full dimensionality. A family of such problems is mixed integer knapsack problems. These problems only have inequality constraints and more details about them can be found in [1]. In [9] many instances of this kind are presented which are of different sizes. From each size there are 5 instances among which we picked one and tried to find integer feasible solutions for them using Randomized Rounding. Moreover we look at the MIPLIB2003 library [7] too, and we try to find instances with the property

that the number of equality constraints plus the inequality constraints that are always satisfied at equality is less than 5 percent of the dimension of the problem.

Moreover, like any other primal heuristic algorithm there could be many different critera for finishing the Randomized Rounding algorithm. In these experiments we terminate after finding 100000 randomly generated points.

For modified Randomized Rounding we find at most 1000 extreme points and we generate 1000 random points. The main reason we find much less random points in modified Randomized Rounding is the fact that the amount of computation that is done for each point is a lot higher compared to Randomized Rounding. In modified Randomized Rounding we solve a linear program for each randomly generated point whereas in Randomized Rounding we just try to see whether the point satisfies the constraints of the problem or not.

On the other hand we also did some computations on the modified version of Randomized Rounding. As was mentioned in the previous section, since modified Randomized Rounding needs to solve many linear programs, it would be beneficial to use it just for problems for which the linear programs that we solve are not computationally expensive. One aspect of linear programs that usually correlated to the computational cost of linear program is its number of non-zero elements in its matrix of constraints. Moreover, since in the linear programs we solve at each randomly generated point the integer variables are fixed, the number of continuous variables in the problem is also important in estimating the computational cost of the linear programs we solve. All in all, modified Randomized rounding seems to perform well for problems with not too many non-zero elements in the matrix of constraints and also not too many continuous variables. It appears that for problems with less than 20000 non-zero elements and less than 200 continuous variables modified randomized rounding is able to find good feasible solutions fast.

To test modified Randomized Rounding we picked the problems in MIPLIB2003 which meet these criteria. There are 17 problems with such specifications which we will use for our experiments.

4.2 Runs and Comparisons

All the test are run on a 64bit machine with a 3.16 GHz Intel Xeon CPU and 8 GB of RAM. Also, Cbc 2.3 is the version we use in these computations.

We tested Randomized Rounding and its modified version on the problems mentioned earlier and here we presen the results. First we look at the performance of Randomized Rounding. Here we compare the stand-alone Randomized Rounding code which is independent from Cbc against Cbc with the default settings. In the default settings the following heuristics are on: Feasibility Pump, RINS, Diving, Combine, Rounding, and Greedy. Table 1 shows the results.

Table 1: Comparing Randomized Rounding against default Cbc for mixed integer knapsack problems

Problems	Optimal Solution	Found by RR	Time	Cbc with cutoff	Cbc without cutoff
mik.250-1-50.1	-33641	-24395.31	1.17	0.04	2.66
mik.250-1-75.2	-49087	-10621.88	1.19	0.91	110.21
mik.250-1-100.3	-69517	-43467.11	1.12	102.14	238.32
mik.250-5-50.4	-33732	-12497.22	1.28	0.46	3.69
mik.250-5-75.5	-50490	-47765.48	1.36	7.95	25.47
mik.250-5-100.1	-67718	-63525.81	1.24	15.71	2188.12
mik.250-10-50.2	-33120	-30216.71	1.28	0.71	10.82
mik.250-10-75.3	-51762	-39628.06	1.41	3.92	14.3
mik.250-10-100.4	-71731	-63303.10	1.23	2.94	7.82
mik.250-20-50.5	-33688	-24528.05	1.29	0.59	2.12
mik.250-20-75.1	-49716	-12933.08	1.39	4.49	30.23
mik.250-20-100.2	-69114	-25131.01	1.06	14.48	50.41
mik.500-1-50.3	-31952	158.76	2.26	6.66	6.76
mik.500-1-75.4	-51761	-32239.87	2.54	22.97	>1800
mik.500-1-100.5	-65587	-8725.78	2.71	670.56	>1800
mik.500-5-50.1	-33738	-29605.40	2.27	0.54	3.23
mik.500-5-75.2	-49391	-46053.99	2.86	3.73	115.5
mik.500-5-100.3	-70724	-67546.09	2.9	3.8	23.65
mik.500-10-50.4	-34046	-12212.97	2.43	1.76	17.14
mik.500-10-75.5	-50547	-39812.07	3	40.89	533.76
mik.500-10-100.1	-69264	-56690.71	3.13	39.65	835.7
mik.500-20-50.2	-33135	-8897.45	2.5	3.02	15.42
mik.500-20-75.3	-51158	-9277.05	2.83	131.6	926.71
mik.500-20-100.4	-72407	-29275.05	3.16	37.01	804.07

The first column of the table is the name of the problems. The second column, "Optimal Solution", is the optimal solution of the problem. The third column, "Found by RR", is the objective value of the feasible solution found by Randomized Rounding. The fourth column, "Time", is the time it took for Randomized Rounding to find the feasible solution. The fifthe column, "Cbc with cutoff", is the time it takes for Cbc to solve the problem having the objective value of the feasible solution found by Randomized Rounding as a cutoff value. And, finally, the last column , "Cbc without cutoff", is the time it takes for Cbc itself with the default settings to solve the problem.

As one can see, in almost all of the instances (except for mik.500-1-50.3) Randomized Rounding is able to find a good feasible solution and adding the objective value of this feasible solution to Cbc as a cutoff value improves the performance of Cbc significantly. Another important fact that we can observe from the table is that Randomized Rounding finds is very fast and it is all due to the fact that Randomized Rounding is a very simple heuristic and there is not any complicated computations involved in it.

Also we tried Randomized Rounding on the MIPLIB2003 instances that are close to full dimensionality and are. Table 2 shows the results of these runs. As one can see Randomized Rounding can not find a feasible solution for a few of these problems. But for the others it finds a good solution fast. For some of these problems Randomized Rounding finds a reasonably close to optimal solution in a fraction of a second.

Table 2: Result of running Randomized Rounding for MIPLIP2003 instances which are close to full dimensionality

Problems	NumCol	NumRow	NumEq	NumSatEq	Opt	Found by RR	Time
aflow40b	2728	1442	78	0	1168	none	-
arki001	1388	1048	20	25	7.58081e+06	none	-
cap6000	6000	2176	123	146	-2.45138e+06	-2440703	3.17
harp2	2993	112	73	10	-7.38998e+07	-61325534	0.91
liu	1156	2178	0	0	?	none	-
manna81	3321	6480	0	0	-13164	-13139	2.5
mas74	151	13	0	0	11801.2	14427.14	0.39
mas76	151	12	0	0	40005.1	43346.22	0.3
mkc	5325	3411	2	0	-563.846	-138.99	13.17
nsrand-ipx	6621	735	0	0	51200	265345.13	5.8
nw04	87482	36	36	0	16862	38818	16.33
opt1217	769	64	48	0	-16	-16	0.35
p2756	2756	755	0	0	3124	none	-

In the last experiment we took some runs to see the performance of modified Randomized Rounding. As mentioned earlier, the problems are taken from MIPLIB2003 library and the results are shown in Table 3. In this table the first five clumns are the name of the problem, number of rows, number of columns, number of non-zero elements in the matrix of constraints, and number of continuous variables in the problem, respectively. The column "Objective" is the optimal objective value of the problem. The column "Feasible" is the feasible solution that modified Randomized Rounding finds, and, finally the last column is the time it takes for modified Randomized Rounding to find that feasible solution. As one can see for most of the problems modified randomized rounding finds a rather good feasible solution fast.

Table 3: Results of runs of modified Randomized Rounding

Name	Rows	Cols	Non-zero	Con	Objective	Feasible	Time
fiber	363	1298	2944	44	405935	8203065.17	0.62
glass4	396	322	1815	20	1.20E+009	none	-
harp2	112	2993	5840	0	-7.39E+007	none	-
liu	2178	1156	10626	67	?	none	-
manna81	6480	3321	12960	0	-13164	-13126	3.53
markshare1	6	62	312	12	1	212	0.33
markshare2	7	74	434	14	1	274	0.51
mas74	13	151	1706	1	11801.2	14776.67	0.32
mas76	12	151	1640	1	40005.1	42218.89	0.52
misc07	212	260	8619	1	2810	none	-
mkc	3411	5325	17038	2	-563.85	-182.44	3.67
modglob	291	422	968	324	2.07E+007	20966737.41	0.05
noswot	182	128	735	28	-41	-40	0.77
opt1217	64	769	1542	1	-16	-16	0.93
p2756	755	2756	8937	0	3124	none	-
pk1	45	86	915	31	11	18	0.08
pp08aCUTS	246	240	839	176	7350	8500	0.1
pp08a	136	240	480	176	7350	none	-

5 Conclusions

There are very few heuristics for constructing initial feasible solutions to mixed-integer programs with general integer variables. We give an opportunistic heuristic (fast but 'dumb') and demonstrate its utility.

References

[1] A. Atamtürk. On the facets of the mixed–integer knapsack polyhedron. *Mathematical Programming*, 98:145–175, 2003.

[2] D. Bertsimas and S. Vempela. Solving Convex Programs by Random Walks. *Jounal of the ACM*, 51(4):540–556, 2004.

[3] E. Rothberg E. Danna and Claude C. Le Pape. Exploring relaxation induced neighborhoods to improve MIP solutions. *Math. Program.*, 102(1(A)):71–90, 2005.

[4] M. Fischetti and A. Lodi. Local Branching. *Mathematical Programming*, 98:23–47, 2003.

[5] M. Fischetti and A. Lodi. Local Branching. *Mathematical Programming*, 98:23–47, 2003.

[6] F. S. Hillier. Efficient heuristic procedures for integer linear programming with an interior. *Operations Research*, 17(4):600–637, 1969.

[7] http://miplib.zib.de/miplib2003.php. .

[8] https://projects.coin or.org/Cbc. .

[9] http://www.ieor.berkeley.edu/ atamturk/data/mixed.integer.knapsack. .

[10] S. Elhedhli J. Naoum-Sawaya. Using analytic centers to find feasible solutions in mixed integer programming. *working paper*, xx:1–24, 2009.

[11] A. Kalai and S. Vempela. Efficient Algorithms for Universal Portfolios. *Journal of Machine Learning Research*, 3:423–440, 2002.

[12] L. Lovasz and M. Simonovits. Random Walks in a Convex Body and an Improved Volume Algorithm. *Random Structures and Algorithms*, 4(4):359–412, 1993.

[13] A. Sinclair M. Jerrum and E. Vigoda. A Polynomial-Time Approximation Algorithm for the Permanent of a Matrix with Nonnegative Entries. *Random Structures and Algorithms*, 4(4):359–412, 1993.

[14] R. Roundy R. Freund and M. Todd. Identifying the set of always-active constraints in a system of linear inequalities by a single linear program. *Working papers from Massachusetts Institute of Technology (MIT), Sloan School of Management*, 104:1674–1685, 1985.

[15] E. Rothberg. An Evolutionary Algorithm for Polishing Mixed Integer Programming Solutions. *INFORM Jounal of Computing*, 19(4):534–541, 2007.