Visualizing Branch-and-Bound Algorithms

Osman Y. Özaltın, Brady Hunsaker
Industrial Engineering Department, University of Pittsburgh, Pittsburgh, PA 15261, USA,
{oyo1@pitt.edu, hunsaker@engr.pitt.edu}

Ted K. Ralphs
Industrial and Systems Engineering, Lehigh University, Bethlehem, PA 18015-1582, USA,
tkralphs@lehigh.edu

We present a suite of tools for visualizing the status and progress of branch-and-bound algorithms for mixed integer programming. By integrating these tools with the open-source codes CBC, SYMPHONY, and GLPK, we demonstrate the potential usefulness of visual representations in helping a user predict future progress of the algorithm or analyzing the algorithm's performance. We have also implemented a flexible toolkit, called the Branch and Cut Analysis Kit, that can be used in conjunction with any instrumented solver to create visual representations in the form of image files. The kit will be made available under an open-source license.

Key words: integer programming; branch-and-bound algorithm
History:

---

1. Introduction

The most often used method for solving mixed integer linear programs (MILP) is the branch-and-bound (B&B) algorithm, which was first proposed by Land and Doig (1960). The effectiveness of this method has substantially increased with recent algorithmic and computational developments. A review of current software systems and their capabilities is contained in Atamturk and Savelsbergh (2005).

In spite of this increased effectiveness, finding a MILP instance for which a B&B algorithm may requires hours or even days to find an optimal solution is not difficult. In such cases, the user is faced with several questions, such as the following: Is it likely that the solution will be proven optimal soon? Is it likely that a better solution will be found soon? What is causing the algorithm to take so long? Would a change in parameters be helpful? Should a reformulation of the model be attempted?

In this paper, we discuss graphical representations of the current state of a B&B algorithm that may help answer question such as those posed above. In contrast to the standard textual output afforded by most MILP solvers, we describe more intuitive methods for assessing progress using graphical displays that can aid the user in recognizing patterns and developing further intuition about the performance of the B&B implementation on a given instance. We describe a toolkit, which we call the Branch and Bound Analysis Kit (BAK), that is very flexible and can easily be interfaced with any solver that can be modified to output status messages in BAK's format. BAK will soon be made available under an open source license for free use.

The paper is organized as follows. A summary of existing visualization approaches for B&B algorithms is given in Section 2. Section 3 presents four different visualization schemes: histograms of LP bounds, scatterplots of LP bounds and integer infeasibilities, incumbent path plots, and B&B tree plots. We discuss each of these schemes in the context of two basic questions: (i) what is the most useful data to visualize and (ii) what is the best way to represent this data? Examples demonstrating potential uses and benefits of each scheme, such as its ability to predict the objective value of the next incumbent solution or to recognize potentially infeasible instances, are presented. Finally, Section 4 contains concluding remarks and possible future research areas.

2. Previous Work

A number of authors have previously suggested various methods for visualizing aspects of the B&B algorithm. One of the very first attempts was by Forrest et al. (1974), who presented an alternative representation of B&B trees that is analogous to the scatterplots of LP bounds and integer infeasibilities presented in Section 3. Nodes in all states (see Table 1) were depicted, and the intent of the image was to motivate the use of best projection for node selection. Some of our B&B tree plots were motivated by those used by Applegate et al. (2006) for display of the B&B trees constructed when solving TSP instances. In their plots, the vertical positions of nodes indicates the magnitude of the associated LP bound. We have incorporated this basic idea into BAK.

We are aware of two existing software toolkits for visualization of B&B algorithms. Leipert (1996) describes VBCTOOL, a graphical interface for visualization of branch-and-cut algorithms. The tree interface of VBCTOOL draws general rooted trees as they are

constructed during branch and bound. It uses shapes and colors to indicate the status of each node and displays information, such as the LP bound, associated with each node. VBCTOOL is capable of interfacing with solvers in a fashion similar to that of BAK. Dash Optimization offers Xpress-IVE (Dash, 2003), a visual development environment for their modeling and optimization suite. It enables the user to analyze and tune the optimization parameters through runtime visualization. More specifically, Xpress-IVE is capable of drawing binary trees for the B&B algorithm.

3. Visual Representations

During execution of the B&B procedure, a great deal of information is available that can be used to assess progress. Commonly used summary statistics displayed as feedback to the user include the number of nodes processed so far, the number of nodes remaining in the queue waiting to be processed, and the current global lower and upper bounds. However, more detailed information is available and includes the following for each node in the tree:

- the optimum objective function value of the node's linear relaxation (its LP bound),
- its position in the tree (e.g., depth and parent node),
- the number of integer infeasible variables, and
- the sum of integer infeasibilities of all variables.

Current methods for monitoring the performance of a B&B algorithm do not generally consider the information listed above. Most implementations show the current incumbent objective value, the MILP gap (the difference between the incumbent and the extreme LP bound), and the number of nodes processed so far. The existing visualization tools mentioned in the previous section also show the parent/child structure of the tree, but they do not show the other information visually.

We present here four visual representations that each display some portion of the available data listed above. These representations are useful because they present the data in a graphical format that is more accessible and easier to process. Our belief is that this will aid users in recognizing trends or patterns in an algorithm's performance. The images could be used both for offline analysis after the solver is finished or in an online fashion while the solver is still running.

We distinguish between several possible states a node in the search tree can be in during the course of the B&B algorithm. These are listed in Table 1. In practice, not all solvers implement all of the states listed in the table. For instance, the three solvers we studied each have either "candidate for processing" nodes or "candidate for branching" nodes but none have both. In the literature, nodes in either of these two states are often referred to as "active" nodes or "live" nodes. For ease of exposition, we will refer to nodes that are candidates for processing or candidates for branching as *candidates*. Most of our visual representations show only candidate nodes.

Table 1: States of nodes during B&B algorithm

branched	feasible non-integer nodes with both children in existence, so no more processing will be necessary.
candidate for processing	nodes having an lp-bound but the bound has not been proven to be optimal yet.
candidate for branching	nodes with a proven lp-bound and having potential children, but not all children exist yet.
fathomed	nodes with an lp-bound worse than the current incumbent.
infeasible	nodes whose linear programming relaxation is proven to be infeasible.
integer	nodes with an integer feasible optimal solution.

3.1. Implementation

The heart of BAK is a Perl script that parses the text output of the solver and extracts information from BAK status messages embedded in that output. After parsing the solver output, BAK prepares data and script files for visual representation and . BAK calls Gnuplot 4.0 (Williams and Kelley, 2004), an open-source graphics package, to generate the plots. The current version of BAK consists of about 1500 lines of Perl code. We modified the source codes of CBC 1.01 (Forrest, 2007), SYMPHONY 5.1 (Ralphs, 2007), and GLPK 4.15 (Makhorin, 2007) so they output the status messages required by BAK. These messages consist of lines of output indicating the creation or change in state (among those given in Table 1) of each node in the search tree. These modifications were relatively simple,

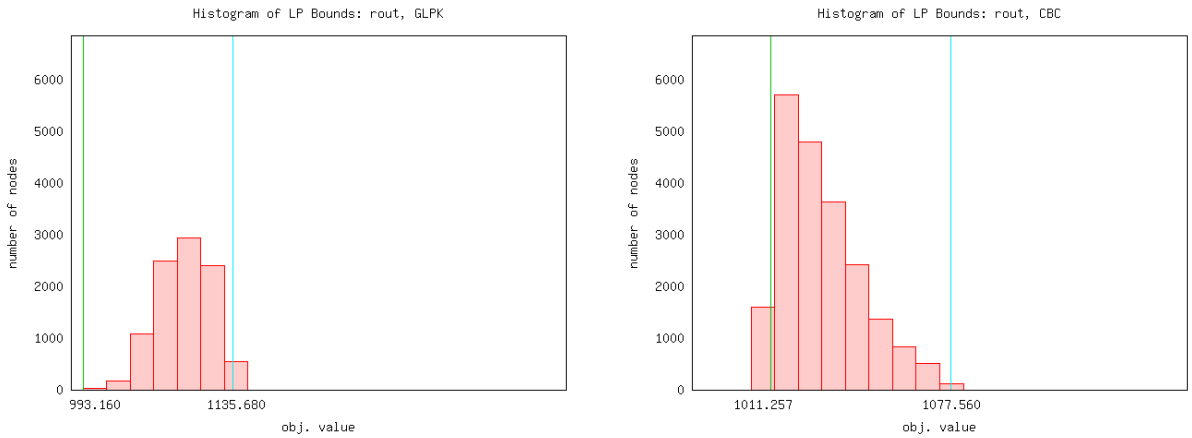

Figure 1: Histograms of instance *rout*, showing the distribution of LP bounds of the candidate nodes (horizontal scales differ)

requiring the addition of just a few lines of code. At present, we assume that B&B trees are binary. This assumption is only necessary for the tree images, however, and it would be relatively straightforward to modify the source code to accommodate solvers that use non-binary branching.

Example instances throughout the paper are taken from the MIPLIB 3.0 (Bixby et al., 1998) and MIPLIB2003 (Achterberg et al., 2006) collections of MILP instances.

3.2. Histograms

Our first visualization technique is to display a histogram of the LP bounds of the candidate nodes in the B&B tree. The best available LP bound for a node is used. If a node has not yet been bounded, then the LP value of the parent node is generally used (though some sophisticated solvers may tighten this bound in various ways).

We generate the histograms by dividing the range of LP bounds into equal intervals, and then counting the number of candidate nodes in each interval. The horizontal axis shows the intervals of LP bounds and the vertical axis shows the number of candidate nodes in each interval. A vertical line on the left hand side of the plots represents the overall LP bound and a vertical line on the right hand side—if present—represents the current incumbent's objective value.

Histograms for the instance *rout* generated after ten minutes of solution time using GLPK and CBC are presented in Figure 1. Note that the horizontal scales are different because the scales were determined based on the entire sequence of images produced by the two solvers.

We see that the overall LP bound is higher and the current incumbent's objective function value is smaller in the plot on the right, which is generated for CBC. This tells us that the MILP gap for CBC after ten minutes is smaller than the MILP gap for GLPK.

What makes the histogram plots useful, however, is not their ability to display the MILP gap, but their ability to show the distribution of the LP bounds of *all* candidate nodes. Observe that in the left plot, more than half of the candidate nodes have LP bounds close to the current incumbent's objective function value but in the right plot, the majority of the candidate nodes have an objective function value close to the overall LP bound. As a result, although the overall MILP gap is smaller in the right plot, most of the candidate nodes in the left plot have smaller individual MILP gaps. This observation demonstrates that considering only the MILP gap when drawing conclusions about the progress of B&B algorithms might be misleading.

Note that in Figure 1 the candidate nodes have LP bounds that appear "well distributed" across the entire range from lower to upper bound. This is not always the case. For instance, in Figure 2, we display the histograms associated with the *mkc* and *10teams* instances generated after 65 and 99 seconds, respectively, using SYMPHONY. Here, we see that candidate

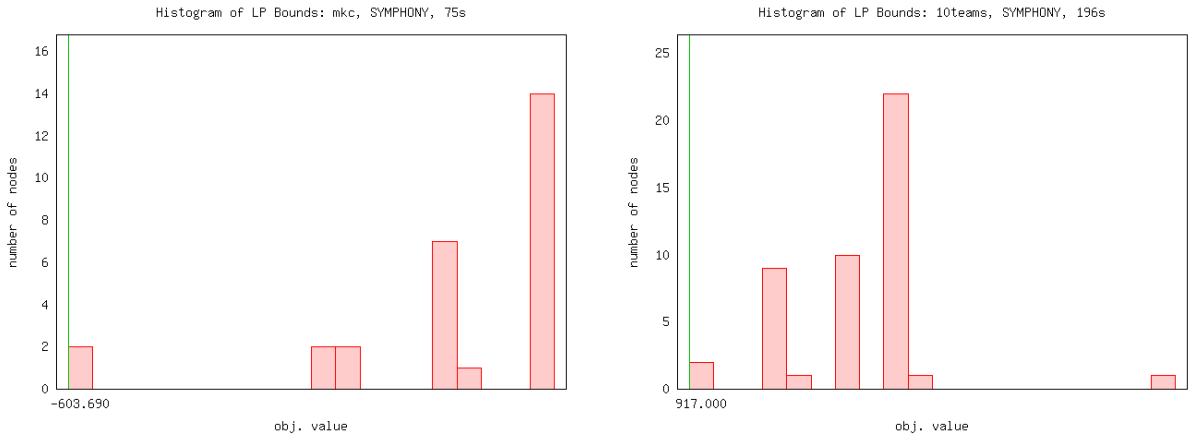

Figure 2: Histograms of instances *mkc* and *10teams*, showing the gaps in the distribution of LP bounds

nodes are accumulated into groups having largely different LP bounds. We conclude that the histogram of LP bounds may help make more accurate observations about the performance of the B&B algorithm on an instance by allowing us to see the complete distribution of LP bounds of the candidate nodes, rather than restricting our view to just the nodes with the highest and lowest bounds.

3.3. Scatterplots

Our second visualization technique is to display a scatterplot of ordered pairs consisting of the LP bound and sum of integer infeasibilities for each node. The plot displays a small red "x" at the location of this ordered pair for each candidate node. In addition, the horizontal line on each plot represents the current incumbent's objective function value. The instance *fiber* is a minimization problem, so each node must have an LP bound greater than or equal to that of its parent. As a result, each node is positioned below and to the right of its descendants in the scatterplots. A node in which an integer feasible solution has been discovered has a sum of integer infeasibilities equal to zero and will therefore appear somewhere on the vertical axis. Our main motivations for these plots are the following two questions: (i) how likely is it that a better incumbent solution will be found? and (ii) how much better will the new incumbent solution be? We demonstrate how the scatterplots can help us answer these two questions by presenting example plots of several instances.

Our first set of example plots is from the instance *fiber* and is given in Figure 3. This instance has 363 rows and 1298 variables. In the left scatterplot which is taken after 3

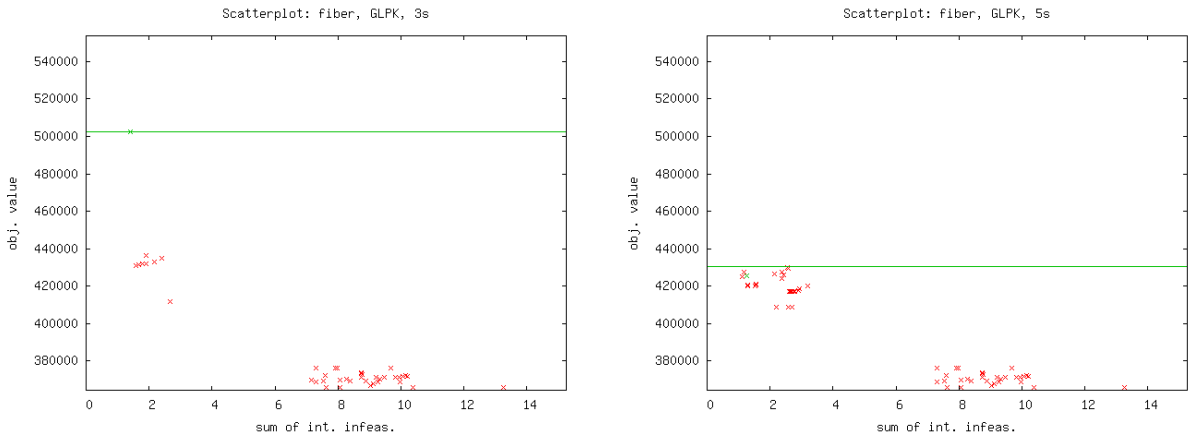

Figure 3: Scatterplots of instance *fiber*, showing the possibility of predicting the objective function value of the next incumbent solution

seconds of solution time, there is one set of candidate nodes for which the sum of integer infeasibilities is approximately 2. The LP bounds of these nodes are between 400,000 and 440,000. Another set of candidate nodes has sum of integer infeasibilities between 7 and 11. This second set of nodes has smaller LP bounds and higher sums of integer infeasibilities. Intuitively, we may conclude that it is more likely we will find an integer feasible solution

with an objective function value close to the LP bounds of nodes in the first set, since those nodes have smaller sums of integer infeasibilities. It seems less likely that there will be an integer solution with an objective function value close to the LP bounds of nodes in the second set of nodes at the bottom of the scatterplots. Looking at the right scatterplot, taken later in the solution process, we see that the intuition was correct in this particular case, as a new incumbent was found with an objective function value around 430,000. The optimum objective value of this instance turned out to be 405,935.18, which is well below the LP bounds of the nodes in the second group.

In addition to guiding the user's intuition, scatterplots might also be helpful in identifying patterns and trends. The set of scatterplots generated for the instance *liu* solved with GLPK are presented in Figure 4. The instance *liu* is known to be a very difficult mixed-integer

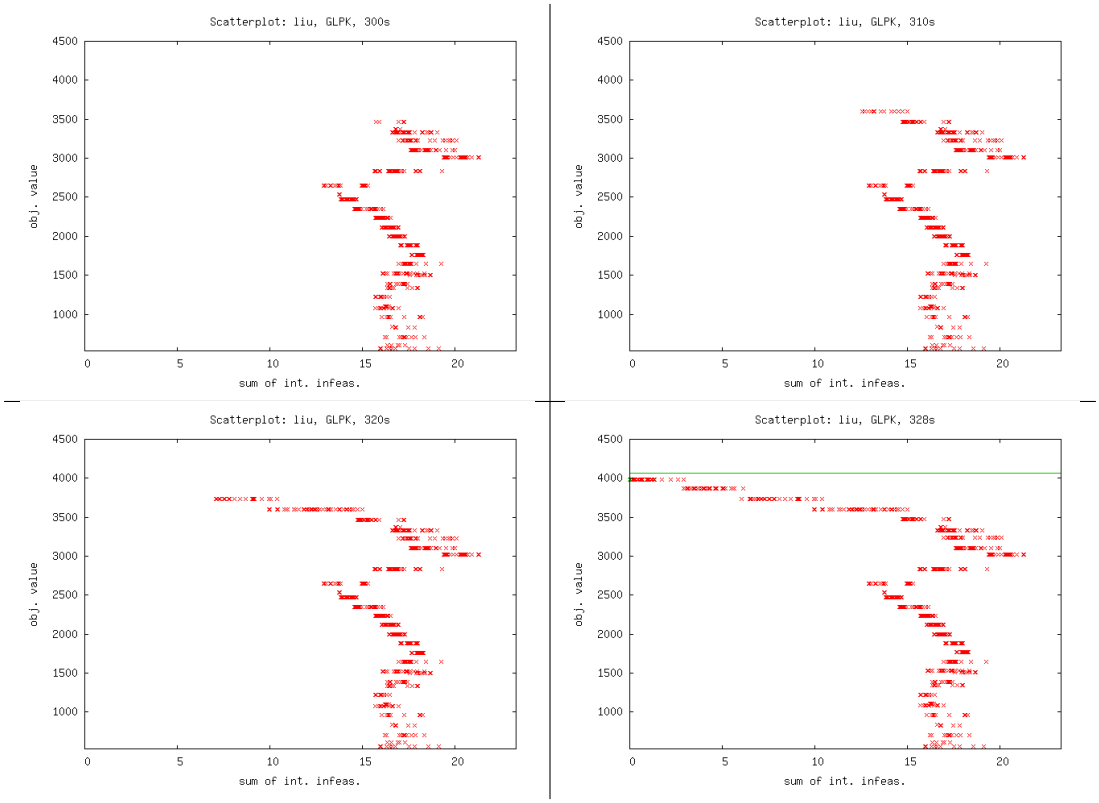

Figure 4: Scatterplots of instance *liu*, showing trends in the LP bound and sum of integer infeasibilities

program with 1089 binary and 67 continuous variables. The scatterplot on the upper left corner of Figure 4 was generated after 300 seconds of CPU time. Looking at this plot, we see that the sum of integer infeasibilities for most of the candidate nodes is between 15 and

20. Considering the number of binary variables, this value is not very large. Nevertheless, it still does not appear that an integer feasible solution will be forthcoming anytime soon, since the plot on the upper right corner, taken later in the solution process, is not much different from the first one, although we do see a few candidate nodes with smaller sums of integer infeasibilities appearing near the top. In the lower left plot, there are more nodes with small sums of integer infeasibilities and it now appears that there is a strong trend towards zero integer infeasibilities.

Note that the LP bounds of the candidate nodes with smaller sums of integer infeasibilities have a layered pattern. For example, the LP bounds of all candidate nodes with a sum of integer infeasibilities between 10 and 15 are all equal to 3602, while the LP bounds of candidate nodes with a sum of integer infeasibilities between 5 and 10 are all equal to 3734. Assuming that the trend towards a sum of integer infeasibilities equal to zero continues, we can expect to get an integer feasible solution soon with an objective function value higher than 3784. In fact, GLPK finds the first integer feasible solution of value 4068 after 328 seconds of CPU time. This is depicted in the scatterplot on the lower right corner, consistent with the trend observed in the preceding plots.

Another situation in which scatterplots are helpful in recognizing patterns is demonstrated with plots from instance *mod008*. Scatterplots generated from that instance while solving with SYMPHONY are given in Figure 5. Instance *mod008* is a fairly small instance

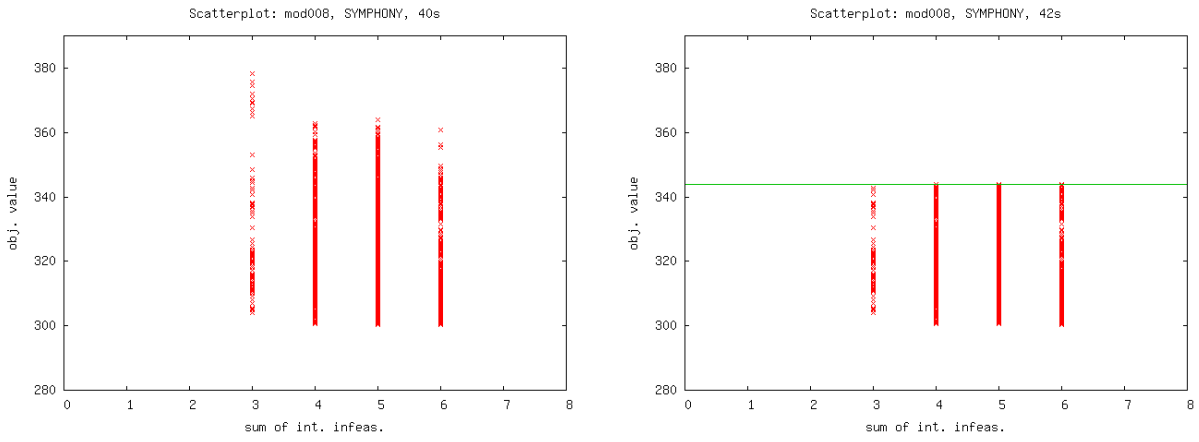

Figure 5: Scatterplots of instance *mod008*, showing the pattern of sums of integer infeasibilities

with 6 rows and 319 variables. From the scatterplots in Figure 5, we see that the sums of integer infeasibilities of each active node is an integer between 3 and 6. One possible expla-

nation for such a pattern is that variables may be taking regular fractional values like 1/4, 1/3 or 1/2 and their sum of integer infeasibilities always add up to whole numbers. Another possible explanation is that there may be pairs of variables conflicting with each other in terms of integer infeasibility. For example, if one of the binary variables takes the fractional value 0.6, another binary variable may be forced to take the fractional value of $1 - 0.6 = 0.4$. That way, their sum of integer infeasibilities always adds up to 1.

The previous examples demonstrate ways that scatterplots might be useful for gaining insight about a problem's structure. Another way that scatterplots can be helpful is in recognizing automatic changes in the behavior of the B&B algorithm. These kinds of adaptive changes are implemented in many advanced MILP solvers. For example, the default node selection rule of CBC is a complicated heuristic method based on best projection. However, that default node selection rule is changed to "best bound" if the number of nodes becomes too high (more than 10,000 nodes). This kind of change is usually not reported, since it is not be of interest to the casual user. Researchers and advanced users, however, might like to know about such changes and their effects. We should point out that even if these changes are reported, tracking their effects in detail would be difficult without the scatterplots.

The scatterplots given in Figure 6 demonstrate this change in solver behavior. These plots were generated consecutively over 10-second time intervals during solution of the instance *qiu* with CBC. Note that CBC changes its default node selection strategy to "best bound" after 352 seconds of CPU time. At that time, the scatterplot of the candidate nodes in the search tree is depicted in the first plot of Figure 6, which is in the upper left corner. In the remaining scatterplots, the candidate nodes at the bottom of the plots start disappearing since they are selected to be processed by the best bound node selection rule. This change does not necessarily help us to find a better incumbent solution immediately, but it reduces the MILP gap by tightening the overall LP bound.

3.4. Incumbent path plots

Our third visualization scheme is the incumbent path plot, which is similar to the scatterplot, except that it displays only the node that produced the current incumbent, along with all its ancestors in the search tree. Such a plot may provide further intuition about the sequence of nodes that lead to a particular feasible solution. We present two examples of incumbent path plots in Figure 7. They were generated for two different incumbents produced during solution of the instance *nw04* by GLPK.

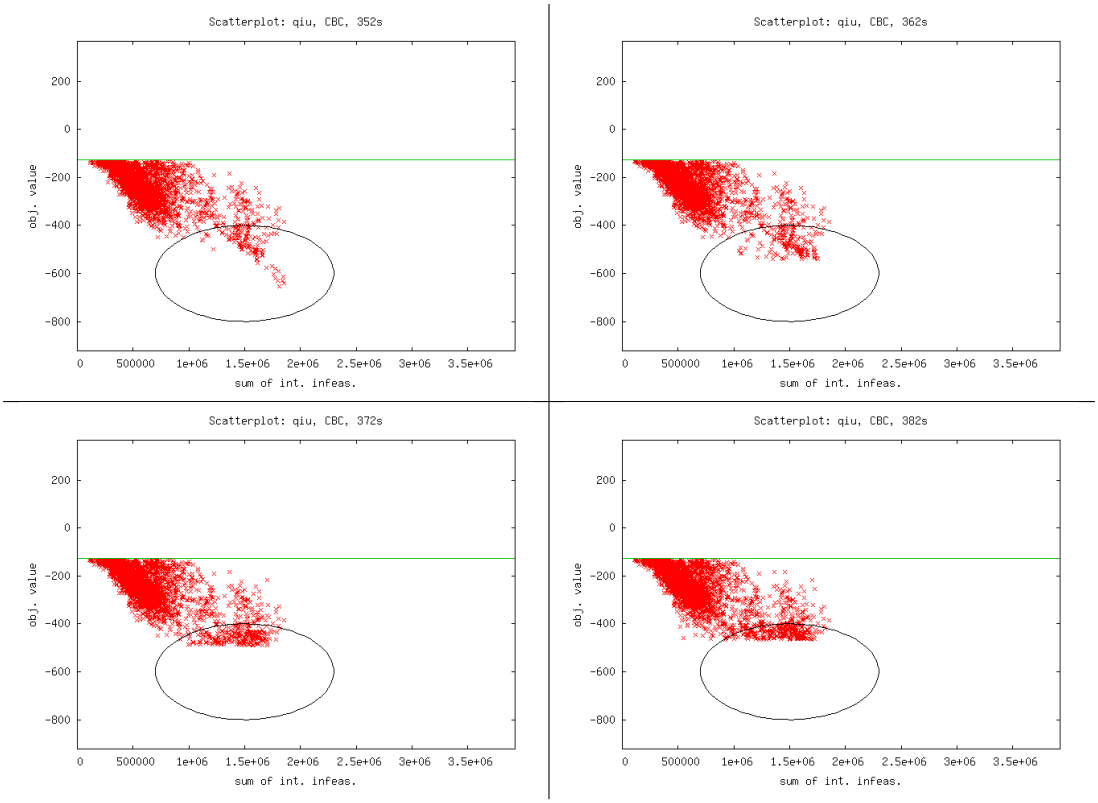

Figure 6: Scatterplots of instance *qiu*, showing a change in the node selection rule

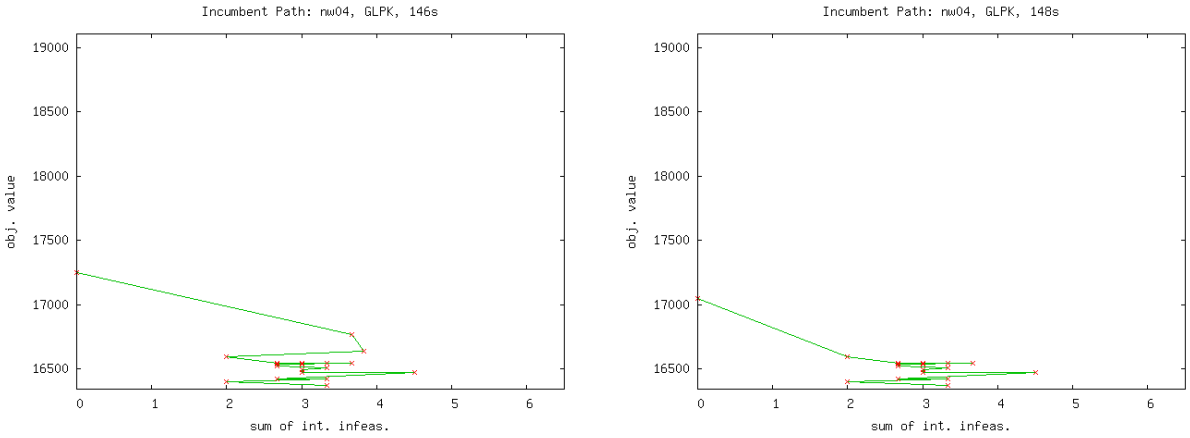

Figure 7: Incumbent path plots of instance *nw04*, an example where the best projection node selection rule doesn't work well

The node with the minimum objective function value, which is located at the bottom of the incumbent path plots, is the root node of the B&B tree, while the node that is plotted on the vertical axis is the one in which the feasible solution was found. The nodes in between these two are ancestors of the node in which the feasible solution was found. Looking at the plots, we see that the ancestors of the two nodes in which feasible solutions were found are mostly the same, or at least have the same associated ordered pairs. This relationship is more apparent from the tree plots of the next section.

Another observation related to the incumbent path plots concerns the node selection rule. Best projection is a widely used node selection rule that makes use of LP bound and integer infeasibility values. The rationale behind best projection is to select the node having the most promising projected objective function value based on its current LP bound and integer infeasibility. From the incumbent path plots in Figure 7 we see that the best projection strategy may not work very well for instance *nw04* since the sums of integer infeasibilities of the ancestor nodes go up and down, preventing any consistent projection of the objective function value.

On the other hand, there are many cases in which the best projection rule works well. For example, consider the incumbent path plots of instance *set1ch* given in Figure 8. There

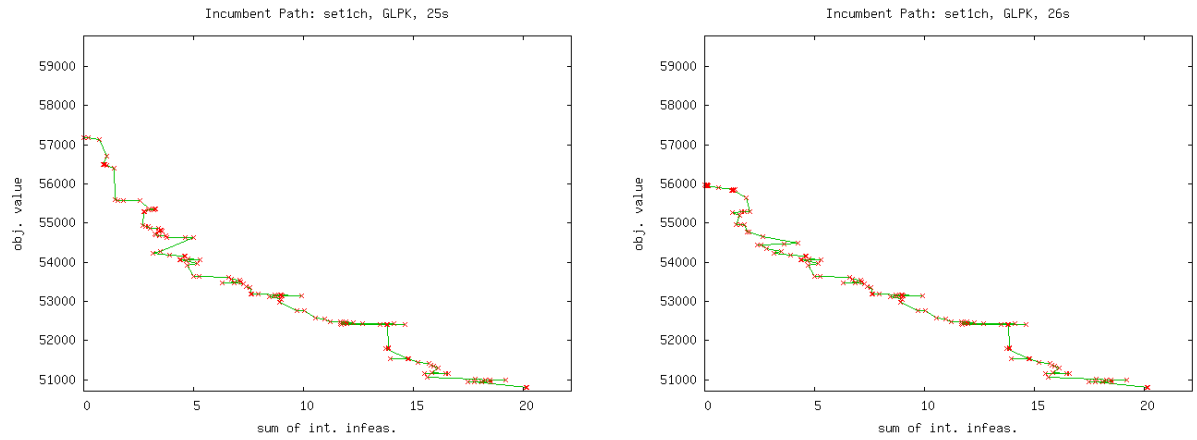

Figure 8: Incumbent path plots of instance *set1ch*, an example where the best projection node selection rule might work well

are again a large number of common ancestors of the incumbent solutions for these two incumbents. In both cases, the rate of increase of the LP bound per unit decrease in the sum of integer infeasibilities remains approximately constant. This indicates that using best projection as a node selection rule might work well for this instance.

3.5. B&B tree plots

Our final visualization technique is to visualize the B&B tree at a fixed point in time in a more traditional fashion, while attempting to incorporate as much additional information about each node in the tree as possible through visual means. These plots are the only ones presented here that show all nodes, regardless of their current state, in the same plot. In these plots, nodes are positioned vertically, according to their LP bounds, with orientation such that the root node is always on top of the plot. There are two options for the horizontal spacing of the nodes. The first one is fixed spacing, which assigns equal space to the children of a node regardless of its number of descendants. The second option is proportional spacing, which assigns horizontal space to each node proportional to its number of descendents. Fixed spacing does a better job of showing whether the tree is balanced, while proportional spacing does a better job of showing all the nodes in the tree. Unless otherwise noted, the B&B tree plots in this paper use the "fixed spacing" option.

Color is one of the most valuable ways of aiding human visualization and plays an especially important role in B&B tree plots. Each node in a B&B tree plot is colored according to its current state (see Table 1). For example, branched nodes are colored in green, infeasible nodes are colored in blue and nodes with an integer feasible solution are colored in red. Of course, for this journal publication, color does not appear. In cases where color would play an important role, we describe the colors in the text.

The first example B&B tree plots were generated while solving instances *rgn* and *mod008* using SYMPHONY and are shown in Figure 9. The root node is located at the top of the tree and branches are plotted down from each node to its children. The horizontal line on each plot represents the current incumbent's objective function value. For a minimization problem, we can avoid processing any node having a lower bound greater than that of the current incumbent, so the nodes under the incumbent line in this figure were either processed prior to the identification of the incumbent or were fathomed by bound prior to processing.

One observation about the plots in Figure 9 is that both of the trees are quite balanced. Another observation concerns the amount of unnecessary work done by the B&B algorithm. By unnecessary work, we mean the processing of nodes which were not necessary to identify and prove the optimal solution. The B&B algorithm may process such nodes if the node selection rule considers them promising before a better integer solution is found. The processing of any node whose parent is under the incumbent line could be considered unnec-

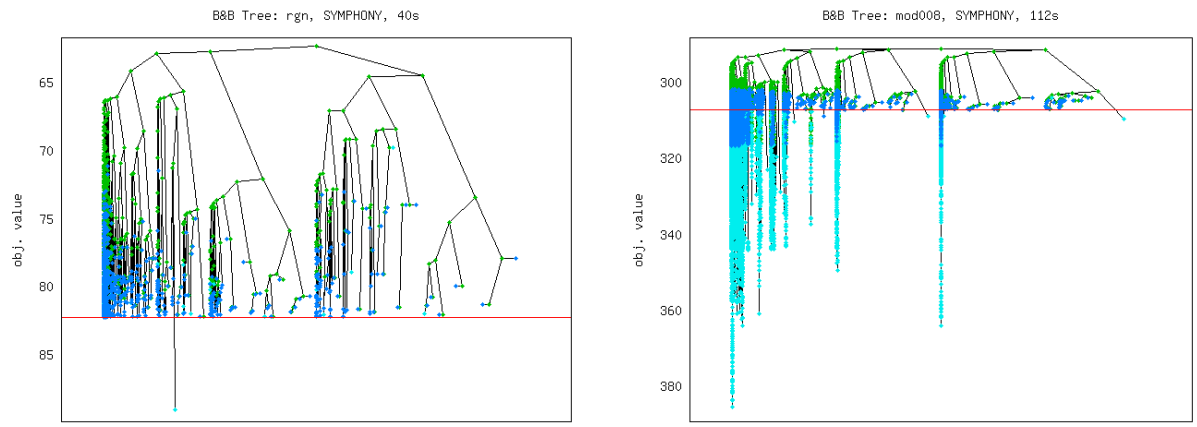

Figure 9: B&B tree plots of instances *rgn* and *mod008*, for comparing the amount of extra work done by the MILP solver

essary, since creation of those nodes could have been avoided either by using a different search strategy or by putting more emphasis on finding an early feasible solution (e.g., through the employment of primal heuristics). It may be that the most efficient solving method does do some "unnecessary" work, but identifying the amount of unnecessary work may be helpful in evaluating an algorithm's performance. Instance *rgn* (on the left) has almost no nodes under the incumbent line, while the majority of the nodes are under the incumbent line for instance *mod008*. This means that the B&B algorithm did a lot of unnecessary work in the case of *mod008*. Obviously, B&B tree plots make it easy to see this difference.

Another possible use of the B&B tree plots is demonstrated by a set of example plots for instance *gesa2_o* given in Figure 10. The first plot depicts the tree generated after 4 hours of CPU time using GLPK and the second plot shows the tree after 91 seconds of CPU time using CBC. If this plot were in color, it would be clear that there are no candidate nodes in the right-hand plot, indicating that CBC reached the optimum solution after 91 seconds. In contrast, there are still many candidate nodes in the left-hand plot, indicating that GLPK hasn't solved the instance after 4 hours of CPU time. There might be several algorithmic reasons for this. What we see from the B&B tree plots is that CBC maintained a much more balanced search tree than GLPK. Actually, the left plot doesn't look much like a tree since GLPK spent almost all of its effort diving into one part of the tree. It is possible that such an unbalanced tree might be a contributing reason for the poor performance of the GLPK on that specific instance, though it is not clear from the plot whether this has only to do with the node selection rule or also with the branching rule.

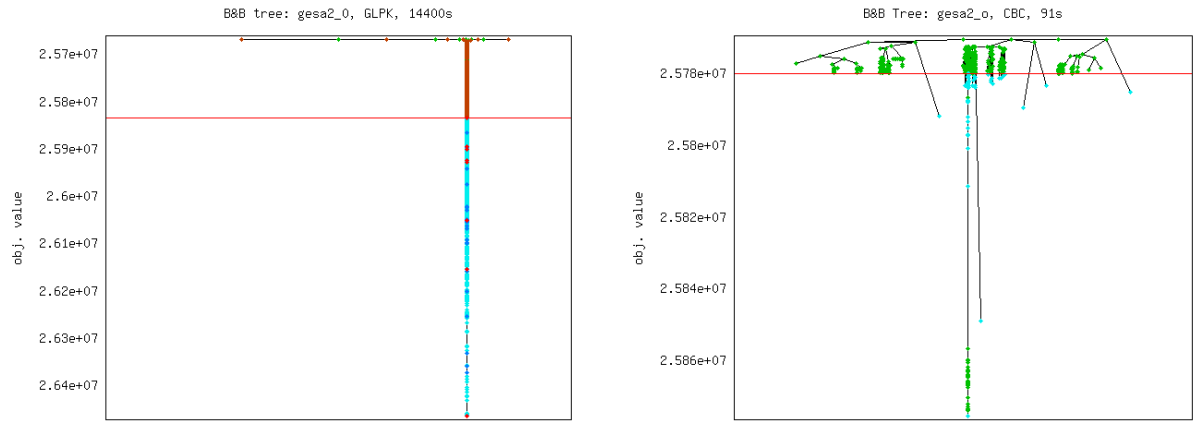

Figure 10: B&B tree plots of instance *gesa2_o*, an example where diving into a branch doesn't work well

On the other hand, diving into a branch and making an unbalanced tree might be helpful in some cases. Consider the plots given in Figure 11 for instance *timtab1*. Although

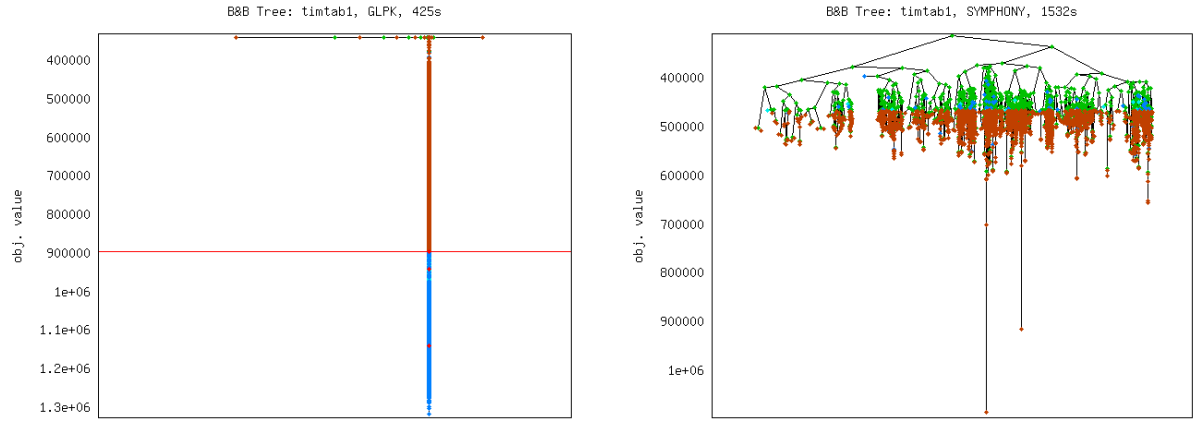

Figure 11: B&B tree plots of instance *timtab1*, an example where diving into a branch works well

SYMPHONY maintained a much more balanced tree than GLPK, GLPK managed to get an incumbent solution after 10 minutes, while SYMPHONY did not. We cannot predict which one of these solvers will find an optimal solution more quickly by considering only the performance in the first ten minutes, but finding an incumbent solution early is often preferable.

Finally, B&B tree plots are particularly helpful for detecting potential infeasibility of an instance, since other types of visual representations don't depict the infeasible nodes at

all. We have encountered some cases where almost all of the candidate nodes in the B&B tree turn out to be infeasible. Such a situation occurs while solving instance *timtab2* using GLPK. The B&B tree plot of that instance generated after 552 seconds of solution time is given in Figure 12. Both of the plots in Figure 12 are of the same tree. However, the left

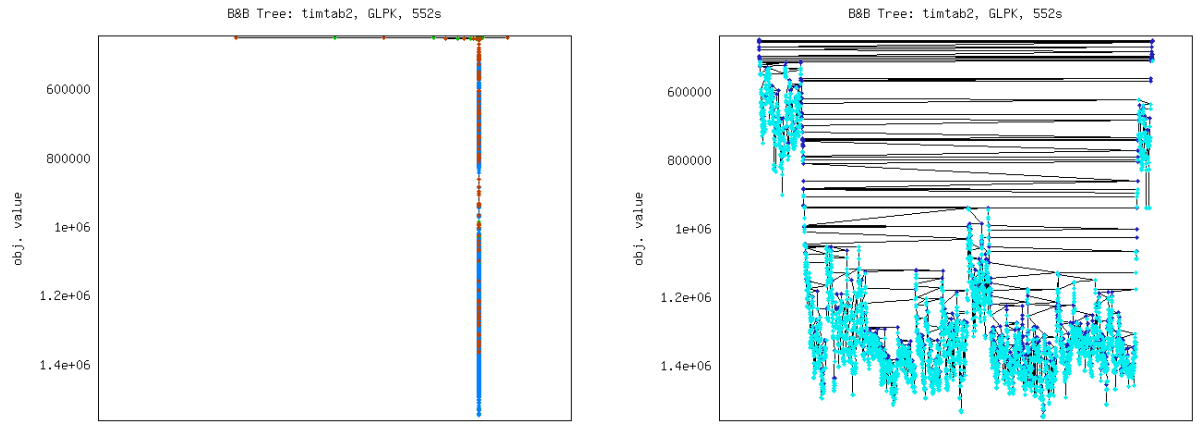

Figure 12: B&B tree plots of instance *timtab2*, showing the large number of infeasible nodes in the tree

plot uses the fixed spacing option while the right plot uses the proportional spacing option. In a color display, almost every leaf node would appear blue, indicating a clear problem with feasibility that would stand in stark contrast to the other B&B tree plots we have considered. Histograms and scatterplots for instance *timtab2* would not reveal this information, since they almost all of the candidate nodes become infeasible right after they created and would not be displayed.

4. Conclusion

We introduced four visual representations to help monitor the progress of B&B algorithms. These representations portray significantly more information available during the B&B algorithm than current methods. We presented examples of how the images may aid users in better understanding solver behavior and in predicting future solver progress on an instance. We hope that this improved understanding may help users decide when to terminate solver runs, when to try different parameter settings, and when to consider reformulating problems.

Generating the images can be accomplished without significantly affecting the total solution time. A Perl script, which we call BAK, generates all the plots presented. BAK will

be released to the community under a free and open-source license. BAK may be used with any MILP solver that generates an appropriate data file with information about the B&B tree. We have already implemented this functionality for three open-source MILP solvers: CBC, GLPK, and SYMPHONY.

Our hope is that researchers may also be able to use the insight they gain to develop more formal methods for some of the computational challenges with mixed-integer programming, such as predicting time to solution, predicting the quality of the eventual solution, or adjusting solver parameters during the course of the algorithm.

References

Achterberg, T., T. Koch, A. Martin. 2006. MIPLIB 2003. *Oper. Res. Lett.* **34** 1–12.

Applegate, David L., Robert E. Bixby, Vasek Chvátal, William J. Cook. 2006. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, Princeton, NJ.

Atamturk, Alper, M.W.P. Savelsbergh. 2005. Integer-programming software systems. *Annals of Operations Research* **140** 67–124.

Bixby, R. E., S. Ceria, C. M. McZeal, M. W. P. Savelsbergh. 1998. An updated mixed integer programming library: MIPLIB 3.0. *Optima* **58** 12–15. URL `citeseer.ist.psu.edu/bixby96updated.html`.

Dash. 2003. *Xpress-IVE*. Dash Optimization.

Forrest, J. J. H., J. P. H. Hirst, J. A. Tomlin. 1974. Practical solution of large mixed integer programming problems with Umpire. *Management Science* **20** 736–773.

Forrest, John J. 2007. *COIN-OR Branch and Cut 1.01*. COIN-OR.

Land, A. H., A. G. Doig. 1960. An automatic method of solving discrete programming problems. *Econometrica* **28** 497–520.

Leipert, Sebastian. 1996. The tree interface – version 1.0 user manual. Tech. Rep. 242, Zentrum für Angewandte Informatik Köln, Lehrstuhl Jünger.

Makhorin, Andrew. 2007. *GNU Linear Programming Kit 4.15*. GNU Project.

Ralphs, Ted K. 2007. *SYMPHONY 5.1*. COIN-OR.

Williams, Thomas, Colin Kelley. 2004. *Gnuplot 4.0*.