Stanford MS&E 236 / CS 225: Lecture 12
GNNs for integer programming

Ellen Vitercik[*]
vitercik@stanford.edu

May 20, 2024

In these notes, we will cover one of the most influential ways that machine learning has been integrated into the branch-and-bound algorithm for integer programming [5]. As in the previous lecture, we will stick with binary integer programs of the form

$$\begin{array}{ll} \text{maximize} & \boldsymbol{c}^T \boldsymbol{x} \\ \text{subject to} & A\boldsymbol{x} \leq \boldsymbol{b} \\ & x_i \in \{0,1\} \quad \text{for all } i \in [n]. \end{array} \tag{1}$$

(However, the methods we will discuss also apply to binary integer programs with continuous variables.) As we discussed in the last class, the key idea of the branch-and-bound algorithm is to intelligently search through an enumeration tree. Each leaf of the enumeration tree corresponds to a solution to the integer program (though it may not be feasible). Using LP relaxations, we can bound the optimal solutions in subtrees of the enumeration tree. We can eliminate entire subtrees when we recognize they only contain solutions with low objective values.

As should be clear from the last class, we must make many different design choices when implementing B&B, which impact its runtime. In these notes, we will focus on a key design choice: variable selection, which can have an enormous impact on B&B's runtime.

1 Variable selection

In Figure 1, we started by branching on variable x_1 in the first layer, then x_2 to create the second layer, and so on. Branching on variables in this fixed, lexicographic way is quite arbitrary, and it's certainly not permutation invariant. We will begin by discussing classical variable selection policies that do not depend on learning and then move on to learning-based variable selection policies.

1.1 Variable selection without machine learning

To build intuition, we will begin with variable selection at the root node, where in Figure 1, we branched on variable x_1. Let $\boldsymbol{x}(1) = (x(1)_1, \ldots, x(1)_n)$ be the optimal solution to the

[*]These notes are course material and have not undergone formal peer review. Please feel free to send me any typos or comments.

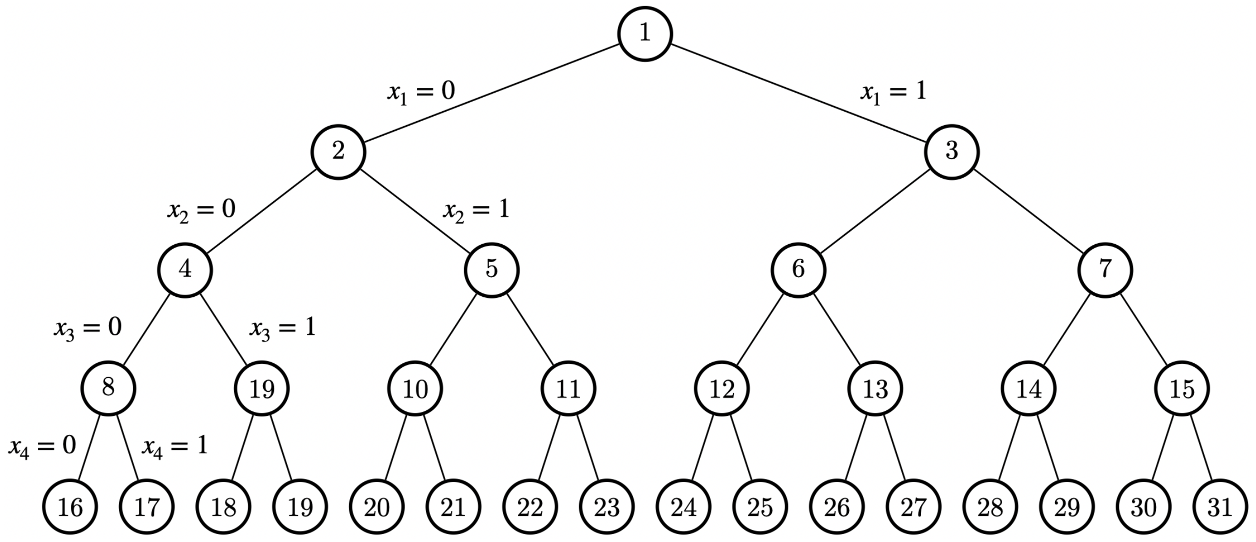

Figure 1: Enumeration tree for an integer program with four binary variables. The leaf labeled 16, for example, corresponds to the solution $\boldsymbol{x} = \boldsymbol{0}$, and the leaf labeled 26 corresponds to the solution $\boldsymbol{x} = (1, 0, 1, 0)$.

input integer program's LP relaxation:

$$\begin{array}{ll} \text{maximize} & \boldsymbol{c}^T\boldsymbol{x} \\ \text{subject to} & A\boldsymbol{x} \le \boldsymbol{b} \\ & x_i \in [0,1] \quad \text{for all } i \in [n]. \end{array} \tag{2}$$

Moreover, let $z = \boldsymbol{c}^T\boldsymbol{x}(1)$ be its objective value.

One of the simplest variable selection policies is the *most fractional variable policy*: branch on the variable $\boldsymbol{x}_i$ where $x(1)_i$ is closest to $\frac{1}{2}$. Intuitively, this is the variable we have the most uncertainty about. Although this policy is generally better than choosing variables lexicographically, we can typically explore less of the enumeration tree if we use a one-step-look-ahead approach.

In particular, suppose we compute the LP optimal solutions if we branched on each variable x_i. Specifically, let z_i^- be the objective value of the optimal solution to

$$\begin{array}{ll} \text{maximize} & \boldsymbol{c}^T\boldsymbol{x} \\ \text{subject to} & A\boldsymbol{x} \le \boldsymbol{b} \\ & x_i = 0 \\ & x_j \in [0,1] \quad \text{for all } j \in [n]. \end{array}$$

Similarly, let z_i^+ be the objective value of the optimal solution to

$$\begin{array}{ll} \text{maximize} & \boldsymbol{c}^T\boldsymbol{x} \\ \text{subject to} & A\boldsymbol{x} \le \boldsymbol{b} \\ & x_i = 1 \\ & x_j \in [0,1] \quad \text{for all } j \in [n]. \end{array}$$

As we have discussed in previous classes, $z_i^- < z$ and $z_i^+ < z$. Thus, $z - z_i^-$ and $z - z_i^+$ measure the change in objective value when we branch on x_i. Intuitively, we want our variable selection policy to take into account both $z - z_i^-$ and $z - z_i^+$, but it's not entirely clear how to combine these scores into a single value (I had a paper that thought about this from a theoretical perspective [3]). A popular approach in practice is called *full strong branching (FSB)* with the product scoring rule, a policy that branches on the variable that maximizes

$$\left(z - z_i^-\right)\left(z - z_i^+\right)^{1}. \tag{3}$$

FSB can be generalized to any node j of the enumeration tree. Inductively, suppose we have branched on some set of variables to arrive at node j. Let Problem(j) be the LP relaxation of the original integer program (Equation (2)) with additional constraints corresponding to each of the variables we have branched on[2]. Let $\boldsymbol{x}(j) = (x(j)_1, \dots, x(j)_n)$ be the optimal solution of LP relaxation corresponding to Problem(j) and $z(j) = \boldsymbol{c}^T\boldsymbol{x}(j)$. Let $z(j)_i^+$ (respectively, $z(j)_i^-$) be the optimal value of LP relaxation corresponding to Problem(j) with additional constraint $x_i = 1$ (respectively, $x_i = 0$). At node j, FSB branches on the variable that maximizes $(z(j) - z(j)_i^-)(z(j) - z(j)_i^+)$.

FSB has pros and cons. On one hand, if we use FSB, we typically only need to search through a very small part of the enumeration tree using B&B before we find the optimal solution. On the other hand, FSB is extremely slow since it requires us to solve many LP relaxations at every node. Although LPs are efficiently solvable, FSB is still untenably slow.

In practice, *pseudo-cost branching* is the most common way of quickly approximating FSB. (**Warning:** this paragraph is a simplified description of pseudo-cost branching. See Appendix A for the nitty-gritty details.) Suppose we are trying to decide which variable to branch on at node j, i.e., Problem(j). Let D_i^+ be the average value of $z(j') - z(j')_i^+$ across all nodes j' where we *already* branched on the variable x_i. This means that we already calculated $z(j')_i^+$, so computing D_i^+ has low overhead. Similarly, let D_i^- be the average value of $z(j') - z(j')_i^-$ across all such nodes j'. Intuitively, you can think of pseudo-cost branching as branching on the variable that maximizes $D_i^+ \cdot D_i^-$. Finally, in *reliability pseudo-cost branching*, if a variable has only been branched on a small number of times, and thus D_i^+ and D_i^- are poor estimates, we fall back on FSB to compute Equation (3).

1.2 GNNs for variable selection

A key observation is that the pseudo-cost branching rule is like a naive machine learning technique for quickly *predicting* the decisions of FSB. The approach in this section takes this intuition to the next level and significantly improves over pseudo-cost branching. This approach was introduced by Gasse et al. [5] and builds on prior research by Khalil et al. [8], Alvarez et al. [2], and Hansknecht et al. [7].

Overall, our goal will be to learn a branching policy $\pi_{\boldsymbol{\theta}}$, where $\pi_{\boldsymbol{\theta}}(i \mid s_t)$ is the probability of branching on the variable x_i when the solver is in "state" s_t and $\boldsymbol{\theta}$ represents the ML model's parameters. There are a few questions we must address to achieve this goal.

[1]This is a simplification. This policy actually branches on the variable that maximizes

$$\{z - z_i^-, \gamma\} \cdot \{z - z_i^+, \gamma\}$$

where $\gamma = 10^{-6}$. Comparing $z - z_i^-$ and $z - z_i^+$ to γ allows the algorithm to compare two variables even if $z - z_i^- = 0$ or $z - z_i^+ = 0$. After all, if $z - z_i^- = 0$, then Equation (3) equals 0, canceling out the value of $z - z_i^+$ and thus losing the information encoded by this difference.

[2]For example, in Figure 1, Problem(5) would include the additional constraints that $x_1 = 0$ and $x_2 = 1$.

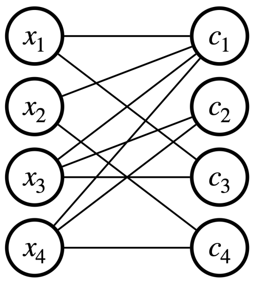

Figure 2: Clause-variable graph for the IP in Equation (4).

Question 1: How should we represent the "state" s_t of the solver? Suppose the solver is deciding which variable to branch on at node j, i.e., Problem(j). The key idea—which will allow us to use GNNs—is to encode Problem(j) as a variable-clause graph with node and edge features. To illustrate, suppose we have the following input IP:

$$\begin{array}{lll} \text{maximize} & 9x_1 + 5x_2 + 6x_3 + 4x_4 & \\ \text{subject to} & 6x_1 + 3x_2 + 5x_3 + 2x_4 \leq 10 & (c_1) \\ & x_3 + x_4 \leq 10 & (c_2) \\ & x_1 + x_3 \leq 0 & (c_3) \\ & -x_2 + x_4 \leq 0 & (c_4) \\ & x_1, \dots, x_4 \in \{0, 1\}. & \end{array} \tag{4}$$

We have labeled the four constraints as c_1, c_2, c_3, and c_4. Its clause-variable graph—which we can think of as our state encoding for Problem(1)—is illustrated in Figure 2. There is one node per variable and one node per clause. There is an edge between node x_i and clause c_j if variable x_i has a non-zero coefficient in clause c_j. The feature of the edge connecting variable x_i and clause c_j is the coefficient of x_i in c_j. For example, the edge between x_1 and c_1 would be 6. An example of a variable node feature would be that variable's coefficient in the objective function. Meanwhile, an example of a constraint node feature would be the constraint's cosine similarity with the objective. For example, this feature for node c_1 would be

$$\frac{9 \cdot 6 + 5 \cdot 3 + 6 \cdot 5 + 4 \cdot 2}{\|(9,5,6,4)\| \cdot \|(6,3,5,2)\|} \approx 0.99.$$

See the paper by Gasse et al. [5] for more node features.

Question 2: How should we define the function $\pi_{\boldsymbol{\theta}}(\cdot \mid s_t)$? Now that we have a graph representation of our state, we use a 2-layer GNN to compute embeddings $\boldsymbol{h}_1, \boldsymbol{h}_2, \dots, \boldsymbol{h}_n$ for each variable. See the paper by Gasse et al. [5] for more details, but it is quite similar to the basic message-passing neural network that we covered in Lecture 6. Given these embeddings, we compute scalars $f(\boldsymbol{h}_1), f(\boldsymbol{h}_2), \dots, f(\boldsymbol{h}_n) \in \mathbb{R}$, where f is a shallow neural network. Finally, we define $\pi_{\boldsymbol{\theta}}(\cdot \mid s_t) = \text{softmax}\left(f(\boldsymbol{h}_1), f(\boldsymbol{h}_2), \dots, f(\boldsymbol{h}_n)\right)$.

Question 3: How should we train $\pi_{\boldsymbol{\theta}}$? Our final missing piece is training $\pi_{\boldsymbol{\theta}}$. Here, we will use *imitation learning*, where our goal will be to train $\pi_{\boldsymbol{\theta}}$ so that it imitates the decisions of FSB. In the training procedure, we run FSB on a training set of integer programs. In doing

Policy	**Time**	**Wins**	**Nodes**
FSB	$17.30 \pm 6.1\%$	0/100	$17 \pm 13.7\%$
Reliability pseudo-cost branching	$8.98 \pm 4.8\%$	0/100	$\mathbf{54} \pm 20.8\%$
TREES [2]	$9.28 \pm 4.9\%$	0/100	$187 \pm 9.4\%$
SVMRANK [8]	$8.10 \pm 3.8\%$	1/100	$165 \pm 8.2\%$
LMART [7]	$7.19 \pm 4.2\%$	14/100	$167 \pm 9.0\%$
GNN [5]	$\mathbf{6.59} \pm 3.1\%$	**85**/100	$134 \pm 7.6\%$

Table 1: Results for set covering instances. The GNN is trained and tested on "easy" instances with 1000 columns and 500 rows.

Policy	**Time**	**Wins**	**Nodes**
FSB	$3600.00 \pm 0.0\%$	0/0	N/A
Reliability pseudo-cost branching	$1677.02 \pm 3.0\%$	4/65	$47299 \pm 4.9\%$
TREES [2]	$2869.21 \pm 3.2\%$	0/35	$59013 \pm 9.3\%$
SVMRANK [8]	$2389.92 \pm 2.3\%$	0/47	$42120 \pm 5.4\%$
LMART [7]	$2165.96 \pm 2.0\%$	0/54	$45319 \pm 3.4\%$
GNN [5]	$\mathbf{1489.91} \pm 3.3\%$	**66**/70	$\mathbf{29981} \pm 4.9\%$

Table 2: Results for set covering instances. The GNN is trained on "easy" instances with 1000 columns and 500 rows and tested on "hard" instances with 1000 columns and 2000 rows.

so, we collect dataset $\{(s_i, i^*)\}_{i=1}^N$, where x_{i^*} is the variable chosen by the "expert policy" FSB. Finally, we optimize $\boldsymbol{\theta}$ so as to minimize the cross-entropy loss

$$-\sum_{i=1}^{N} \log \pi_{\boldsymbol{\theta}}(i^* \mid s_i).$$

Subset of results

This section presents a small sample of the results by Gasse et al. [5]. Table 1 presents results for randomly generated set-covering IPs. The GNN is trained on "easy" instances with 1000 columns and 500 rows. It is also tested on 100 easy instances. We include results for several earlier baselines—TREES [2], SVMRANK [8], and LMART [7]—which also train ML models to imitate strong branching but do not use GNNs. The "time" column reports runtime in seconds with a timeout of 1 hour. The "wins" column reports (number of instances where the policy has the fastest runtime) / (total number of test instances that the policy solved before the 1-hour timeout). Finally, the "nodes" column reports the size of the B&B tree. We can see that although reliability pseudo-cost branching builds smaller trees, the GNN is faster. This is likely because the trees are so small that, in most cases, reliability pseudo-cost branching is roughly equivalent to FSB. In Table 2, we can see that the GNN continues to dominate the baselines when we train on easy instances but tested on "hard" instances with 1000 columns and 2000 rows.

References

[1] Tobias Achterberg, Thorsten Koch, and Alexander Martin. Branching rules revisited. *Operations Research Letters*, 33(1):42–54, January 2005.

[2] Alejandro Marcos Alvarez, Quentin Louveaux, and Louis Wehenkel. A machine learning-based approximation of strong branching. *INFORMS Journal on Computing*, 29(1): 185–195, 2017.

[3] Maria-Florina Balcan, Travis Dick, Tuomas Sandholm, and Ellen Vitercik. Learning to branch: Generalization guarantees and limits of data-independent discretization. *Journal of the ACM*, 71(2), apr 2024.

[4] Michel Bénichou, Jean-Michel Gauthier, Paul Girodet, Gerard Hentges, Gerard Ribière, and O Vincent. Experiments in mixed-integer linear programming. *Mathematical Programming*, 1(1):76–94, 1971.

[5] Maxime Gasse, Didier Chetelat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2019.

[6] J-M Gauthier and Gerard Ribière. Experiments in mixed-integer linear programming using pseudo-costs. *Mathematical Programming*, 12(1):26–47, 1977.

[7] Christoph Hansknecht, Imke Joormann, and Sebastian Stiller. Cuts, primal heuristics, and learning to branch for the time-dependent traveling salesman problem. *arXiv preprint arXiv:1805.01415*, 2018.

[8] Elias Boutros Khalil, Pierre Le Bodic, Le Song, George Nemhauser, and Bistra Dilkina. Learning to branch in mixed integer programming. In *AAAI Conference on Artificial Intelligence*, 2016.

[9] Jeff Linderoth and Martin Savelsbergh. A computational study of search strategies for mixed integer programming. *INFORMS Journal of Computing*, 11(2):173–187, 1999.

A Pseudo-cost branching

To introduce pseudo-cost branching [4, 6, 9], we first define some notation. At a node j, let $f(j)_i^- = x(j)_i - \lfloor x(j)_i \rfloor$ and $f(j)_i^+ = \lceil x(j)_i \rceil - x(j)_i$ denote how far $x(j)_i$ is from being integral. Let $c(j)_i^-$ and $c(j)_i^+$ be the objective gains per unit change in variable x_i at node j after branching in each direction. More formally,

$$c(j)_i^- = \frac{z(j) - z(j)_i^-}{f(j)_i^-} \quad \text{and} \quad c(j)_i^+ = \frac{z(j) - z(j)_i^+}{f(j)_i^+}.$$

Let σ_i^- (respectively, σ_i^+) be the sum of $c(j)_i^-$ (respectively, $c(j)_i^+$) over all nodes j where x_i was chosen as the branching variable. Let η_i be the number of such nodes. The pseudo-costs of variable x_i are defined as

$$D_i^- = \frac{\sigma_i^-}{\eta_i} \quad \text{and} \quad D_i^+ = \frac{\sigma_i^+}{\eta_i}.$$

We initialize the pseudo-costs using strong branching: if $\eta_i = 0$ when we are choosing which variable to branch on at a node j, we set

$$D_i^- = \frac{z(j) - z(j)_i^-}{f(j)_i^-} \tag{5}$$

and similarly for D_i^+.

In pseudo-cost branching, we estimate $z(j) - z(j)_i^-$ using the value $D_i^- f(j)_i^-$ and we estimate $z(j) - z(j)_i^+$ using the value $D_i^+ f(j)_i^+$. For example, Equation (3) would become $\left(D_i^- f(j)_i^-\right) \cdot \left(D_i^+ f(j)_i^+\right)$.

Reliability branching [1] is a variation on pseudo-cost branching. Variable i's pseudo-costs are said to be *unreliable* if $\eta_i < \eta_{\text{rel}}$, where $\eta_{\text{rel}} \in \mathbb{Z}$ is a tunable parameter. We set the pseudo-costs of unreliable variables as in Equation (5).