

# Learning to Branch: New Results for the Quadratic Assignment Problem

Nathan Brixius, June 2023



[bit.ly/kurt-qap](https://bit.ly/kurt-qap)

# Key Points

Branching is a primary performance driver for branch-and-bound algorithms for the Quadratic Assignment Problem (QAP).

Branching rules obtained by learning-to-rank methods can outperform traditional hard-coded rules.

This approach can likely be used for most QAP bounds, and for problems beyond QAP.

# Linear Assignment Problems (LAP)

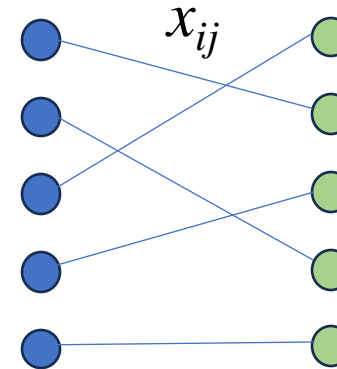
$$\min \sum_{i,j}^n c_{ij} x_{ij}$$

$$\sum_i^n x_{ij} = 1, \quad j = 1..n$$

$$\sum_j^n x_{ij} = 1, \quad i = 1..n$$

$$x_{ij} \in \{0, 1\}$$

$$X \in \Pi$$



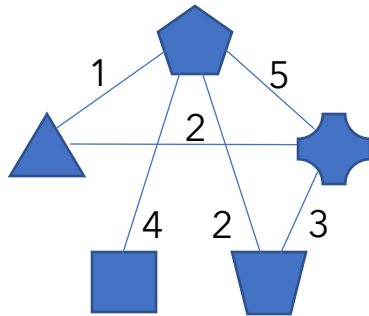
AKA minimum cost weighted bipartite matching

Birkhoff 1946: The permutation matrices constitute the extreme points of the set of doubly stochastic matrices  $\rightarrow$  x integrality can be relaxed  $\rightarrow$  LAP is easy

# Quadratic Assignment Problem (QAP)

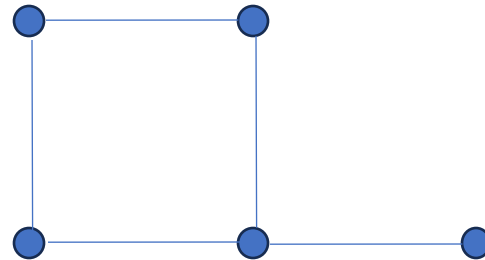
$$\min_{X \in \Pi} \sum_{i,j,k,l}^n a_{ij} b_{kl} x_{ik} x_{lj} + \sum_{i,j}^n c_{ij} x_{ij}$$

flows (A)



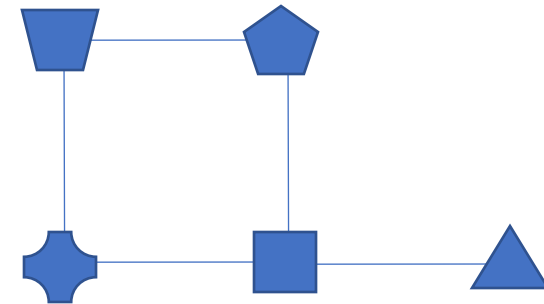
$$\begin{bmatrix} 0 & 5 & 2 & 4 & 1 \\ 5 & 0 & 3 & 0 & 2 \\ 2 & 3 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 \\ 1 & 2 & 0 & 0 & 0 \end{bmatrix}$$

distances (B)



$$\begin{bmatrix} 0 & 1 & 2 & 2 & 3 \\ 1 & 0 & 2 & 1 & 2 \\ 2 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 2 & 0 & 0 & 0 \end{bmatrix}$$

assignment (X)



$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

# Divide and Conquer using Partial Assignments

Let  $Q = (A, B, C)$  be a QAP of size  $n$ .

Assigning  $i \rightarrow j$  yields a QAP  $Q^{ij}$  of size  $n-1$ :

$$Q^{ij} = (A^{ij}, B^{ij}, C^{ij}, d^{ij})$$

$$A^{ij} = A_{(ii)}$$

$$B^{ij} = B_{(jj)}$$

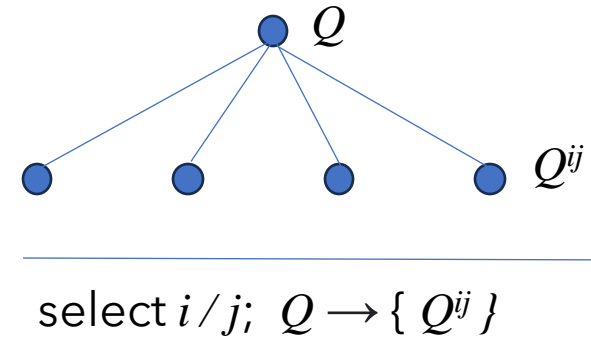
$$C^{ij} = C_{(ij)} + 2\hat{a}_i\hat{b}_j$$

$$d^{ij} = a_{ii}b_{jj} + c_{ij}$$

where

$M_{(ij)}$  =  $M$  with row  $i$  and column  $j$  removed

$\hat{a}_i$  = row  $i$  of  $A$  with  $i$ th element removed

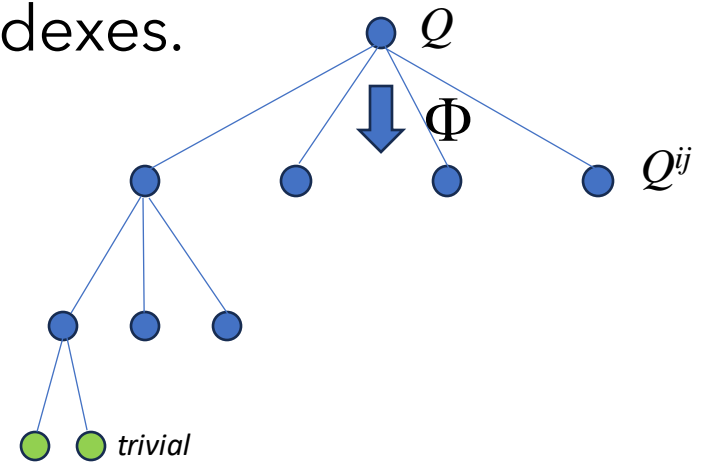


# Solving QAP using Enumeration Trees

Let  $I = I_R \cup I_C$  be the set of row and column indexes.

Selecting  $i \in I$  is called *branching*.

$\Phi(Y) \rightarrow i$  is a *branching method*.

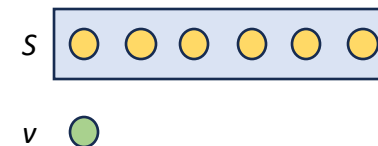
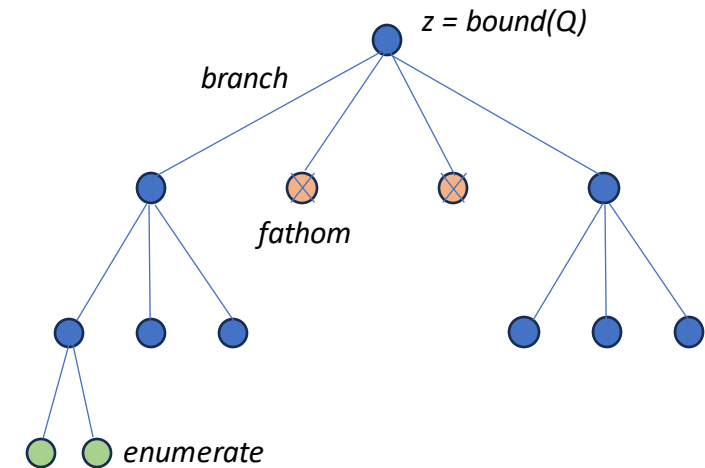


Repeatedly applying  $\Phi$  generates a tree of QAPs, with trivial subproblems at the leaves.

The size of the tree is exponential in  $n$ .

# Branch and Bound: Trimmed Enumeration Trees

```
def branch_and_bound(P):  
    v = initial_solution(P)  
    S = {P}  
  
    while not empty(S):  
        Q = take(S)  
        if easy(Q):  
            v = enumerate(Q, v)  
        else:  
            z, Y = bound(Q) # Y = branching context  
            if z <= v: # no fathom  
                S += branch(Q, Y)  
  
    return v
```



# Branch and Bound: Four Things Matter

Thing	Aspiration	Metric
<b>Bound Quality</b>	Fathom as often as possible	$(z - w) / w$
<b>Bound Speed</b>	Calculate as fast as possible	cpu time
<b>Branching Quality</b>	Make future nodes as easy as possible	??
<b>Branching Speed</b>	Create new nodes as fast as possible	cpu time

Lots of research about this...

...not nearly as much about this



# Quadratic Programming Bound: pretty fast, pretty good

Anstreicher, Brixius (2001):

$$\begin{aligned} \text{QPB :} \quad & \min \mathbf{vec}(X)^T Q \mathbf{vec}(X) + C \bullet X + \langle \lambda(\hat{A}), \lambda(\hat{B}) \rangle_- \\ & \text{s.t. } Xe = X^T e = e \\ & X \geq 0, \end{aligned}$$

For easily computed  $Q, A, B, S, T$ .

Advantages:

---

Good quality

Can solve fast using LAP

Iterative solve  $\rightarrow$  can stop anytime

Yields dual matrix  $U$

---

# QPB vs Gilmore-Lawler (GLB) for nug18 test problem

Total Time: 39.8741 seconds

Nodes: 265917 / 215368 / 0

root bound: 1688.79

fw iterations: 7668323

Total Time: 244.089 seconds

Nodes: 34717569 / 25994653 / 5

root bound: 1554

fw iterations: 0

level	nodes	%fathom	%c.elim	time
0	1	0.0000	0.0000	0.18
1	18	0.0000	0.0131	1.94
2	302	0.0033	0.1186	2.34
3	4245	0.3965	0.3452	3.04
4	25165	0.6500	0.4911	6.43
5	62741	0.7630	0.5821	10.53
6	80792	0.8339	0.6398	9.10
7	57990	0.8739	0.6806	4.56
8	25682	0.9014	0.7156	1.41
9	7202	0.9204	0.7194	0.29
10	1447	0.9109	0.7277	0.05
11	281	0.9253	0.7143	0.01
12	42	0.9048	0.7500	0.00
13	6	0.6667	0.7000	0.00
...				

level	nodes	%fathom	%c.elim	time
0	1	0.0000	0.0000	0.00
1	18	0.0000	0.0000	0.00
2	306	0.0000	0.0000	0.01
3	4896	0.0071	0.0199	0.08
4	71465	0.0649	0.1282	0.99
5	815646	0.2792	0.3498	8.80
6	4969819	0.5441	0.5546	42.93
7	12111005	0.7279	0.6882	88.74
8	11303625	0.8316	0.7669	72.26
9	4435710	0.8849	0.8073	25.25
10	885601	0.9114	0.8269	4.50
11	108631	0.9232	0.8293	0.50
12	9965	0.9264	0.8147	0.04
13	815	0.9276	0.7932	0.00
...				

# Dual branching

Many bounds produce dual matrices  $U$  that are useful for branching.

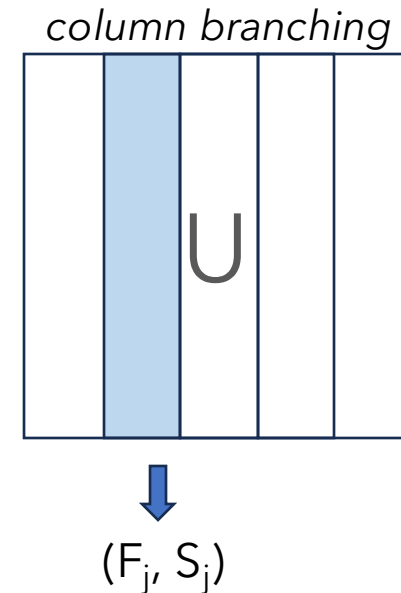
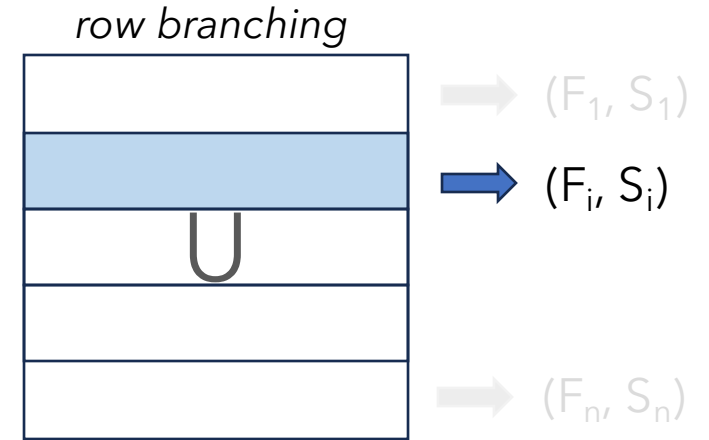
Suppose  $u_{ij} \leq \text{bound}(Q_{ij})$ .

Let's minimize the number of subproblems created, using net bound improvement as a tiebreaker.

$$\Phi(Y) = \arg \text{lexmin}_i (f_i, s_i)$$

$$f_{ij}^r = \begin{cases} 1 & \text{if } z_{ij} \geq \text{bound}(Q_{ij}) + u_{ij} \\ 0 & \text{otherwise} \end{cases}$$

$$s_{ij}^r = \sum_j u_{ij}$$



# Strong branching uses bounds from subproblems

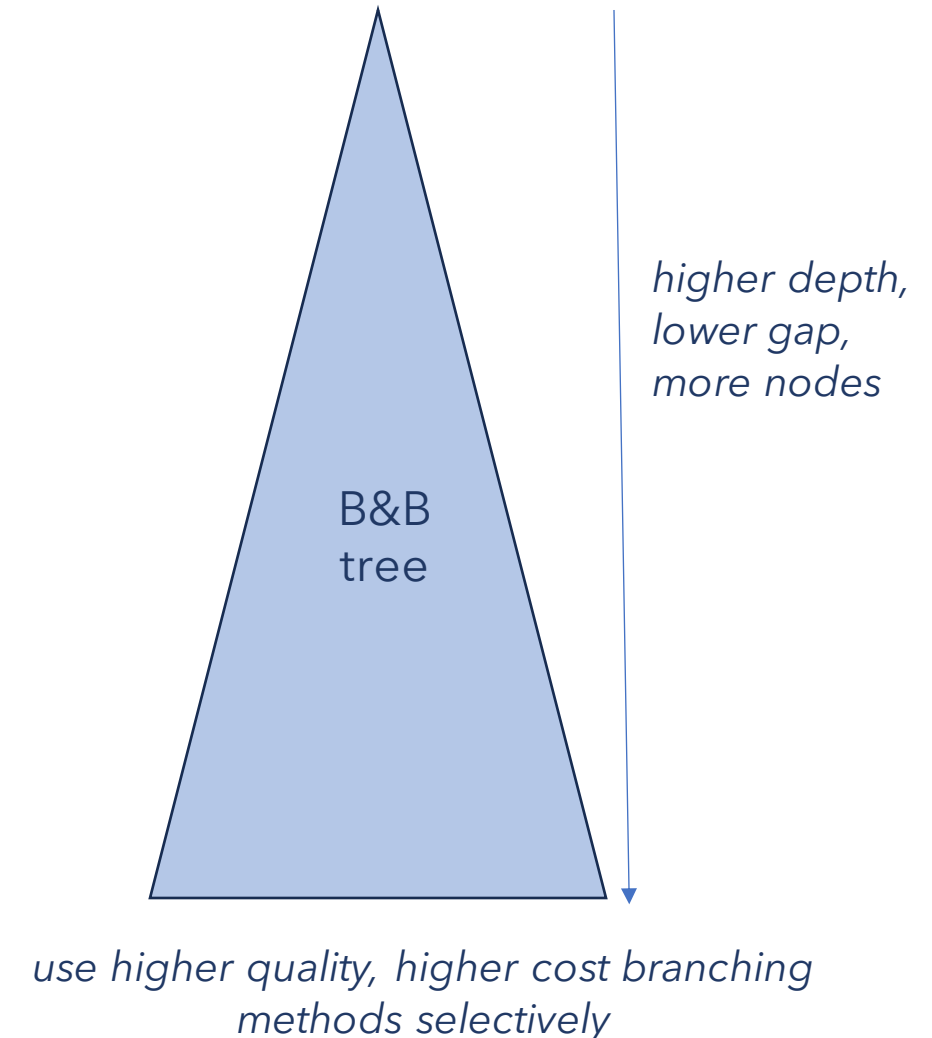
Let  $z_{ij} = \text{bound}(Q_{ij})$

*Strong branching*:  $\Phi$  defined as before, using  $Z$  in place of  $U$ .

Computational cost is  $O(B n^2)$  where  $B$  is the cost of the bound

*Lookahead branching* combines strong branching and dual branching:

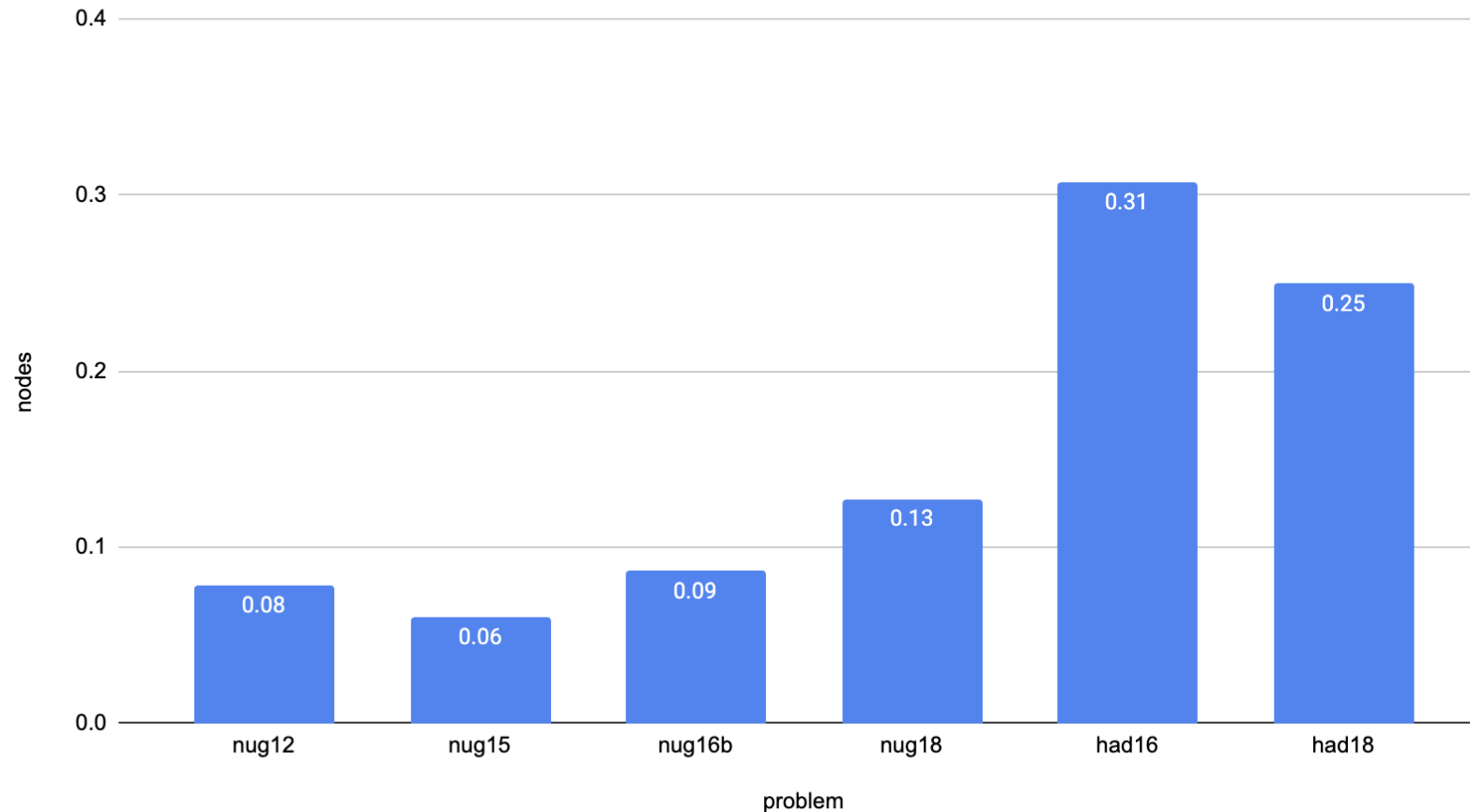
- $Z$  at level  $n+1$
- $U$  at level  $n+2$  (which comes for free when computing  $Z$ )



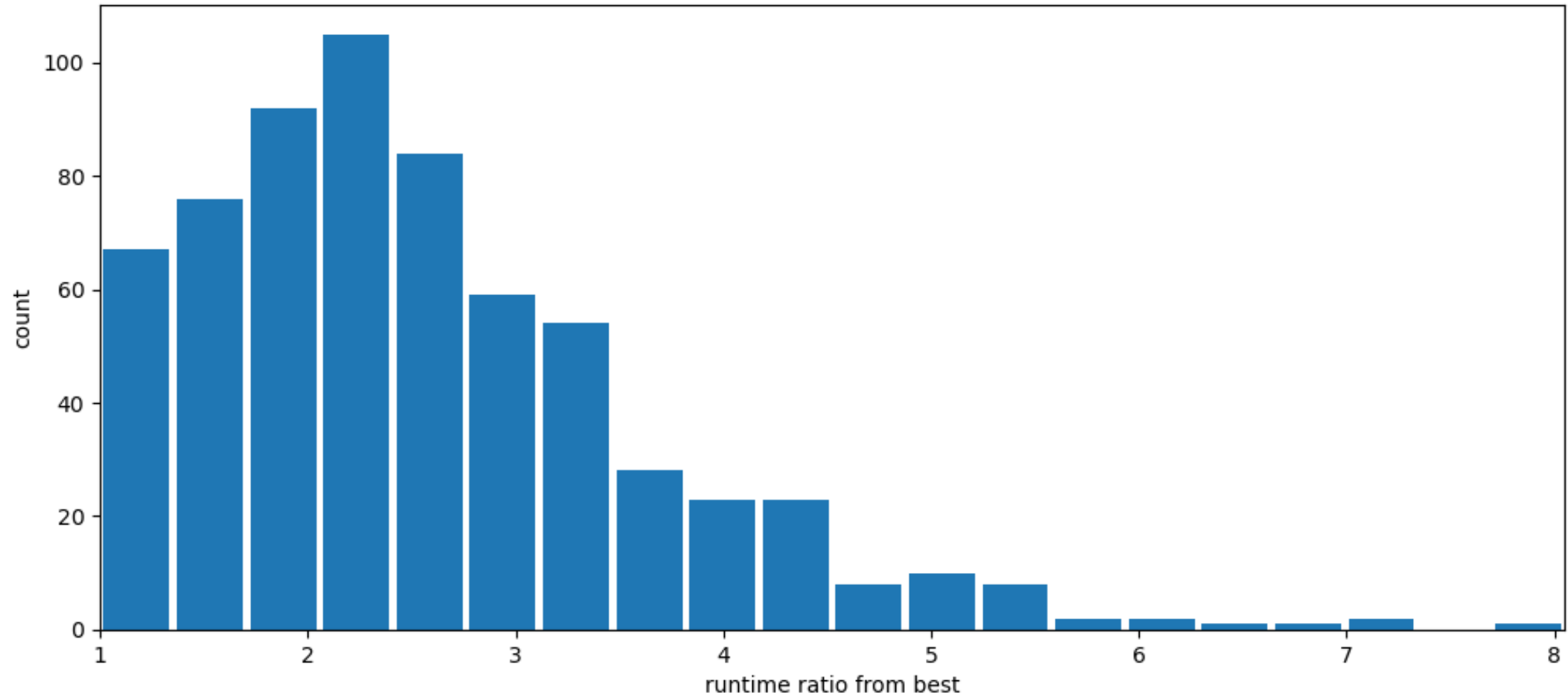
# Strong branching reduces tree size

Reduction in nodes from strong branching increases with problem size

Node count: strong / dual branching



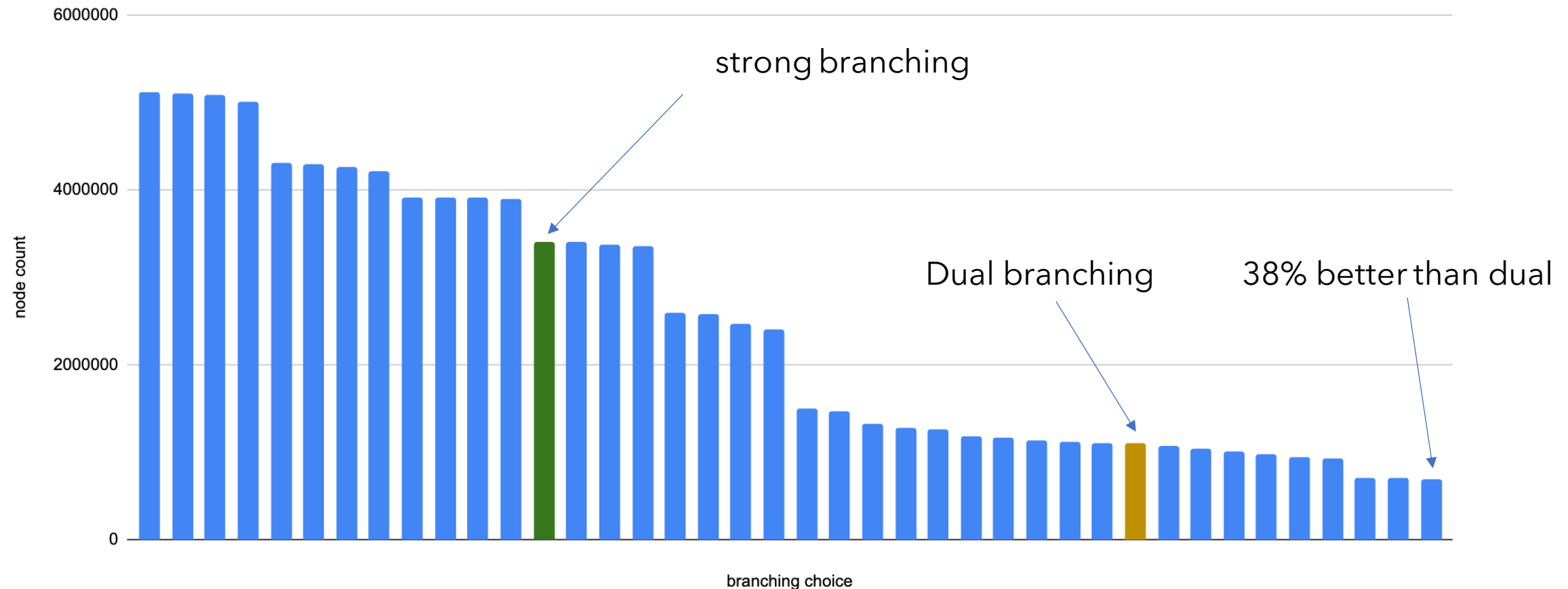
# Variance of 650 branching choices at level 4 of nug16b



Variance from a **single** branching choice

# Branching matters even more for larger problems

nug20 total nodes by branching choice



7.5x between best and worst: one branching choice

# Finding branching rules is a learning to rank problem

What matters is which branching choice is best:

$$\arg \min_{i \in I} \Phi(Y_i)$$

If choice quality is given by a loss function  $L$ :

$$\arg \min_{R(\Phi)} L(\Phi)$$

Minimizing the loss function over the space of ranking models is a **learning-to-rank** problem.

$L$  should involve  $t_i$ , the B&B running time given choice  $i$ .



# Learning-to-rank with Lambdarank

Given  $q$  queries each with

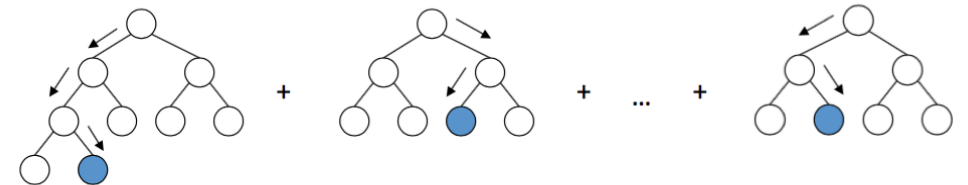
- Features  $F$
- Ground truth relevance rank  $Y$

Want estimated relevance score  $s$ :

$$\min_s \sum_{Y_i > Y_j} \omega(i, j) \log(1 + e^{-\sigma(s_i - s_j)})$$

Lambdarank minimizes this logloss by gradient boosting on  $(F, Y)$ .

Trained model is a set of if-else decision trees on the features  $F$ .



i	s	rank(s)	t
0	0.657	14	157
1	0.631	12	145
2	0.432	3	73
3	0.898	23	267
4	0.123	1	75
..			...
29	0.852	28	311

See:

- ["From RankNet to LambdaRank to LambdaMART: An Overview"](#) Burges (2010)
- ["The inner workings of the lambdarank objective in LightGBM"](#) Fineis (2021)

# Lambda branching: Learning-to-rank in B&B

## Create Training Data

Run B&B and log the following:

- trial, i, X, U, w, z, t

## Prepare Feature Data

F = trial | U

Y = t

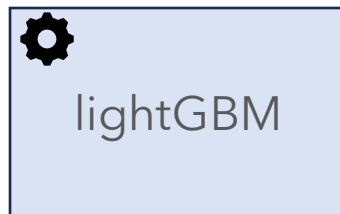
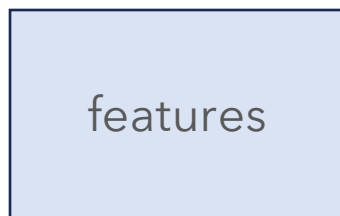
## Train Model

m = lgb.train(X, y, 'lambdarank')

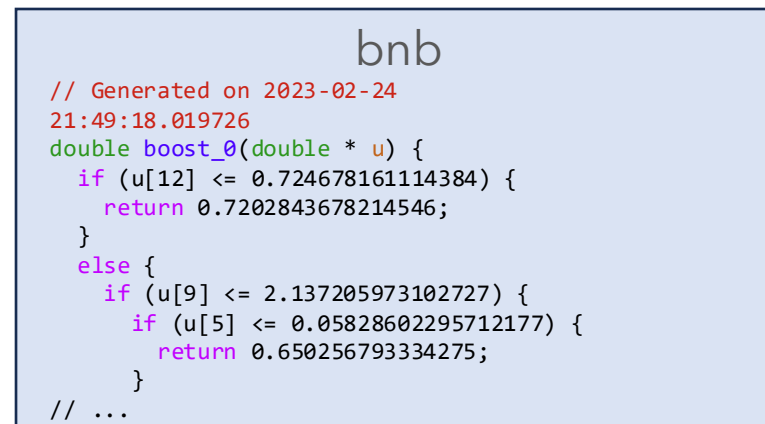
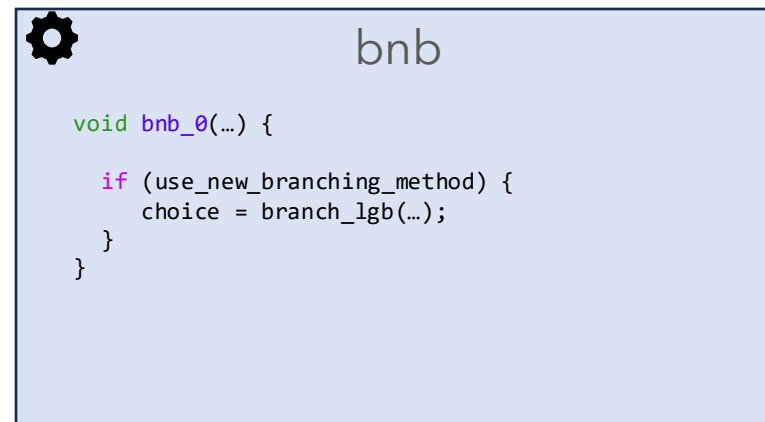
## Generate Branching Code

New branching method "lambda"

Offline

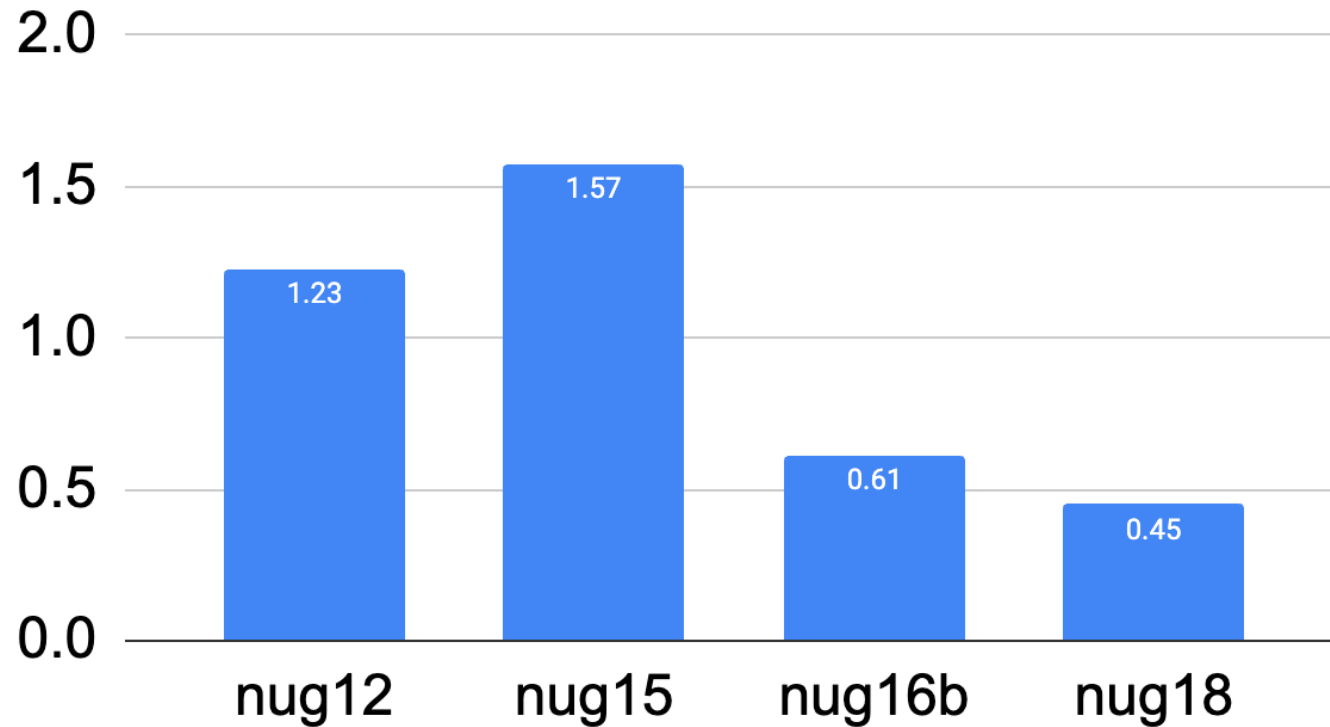


Online



# Lambda compared with dual branching is promising

lambda nodes / dual nodes



problem	dual	lambda
nug12	1746	2145
nug15	9338	14705
nug16b	28318	17365
nug18	869707	391055

# Next Steps and Extensions

Improve utility for QAP

- Use for strong branching
- Feature engineering:  $X, (A, B, C)$ , transforms
- Improved tuning and training

Extending usage

- Learn branching strategy selection rules
- Domains outside QAP