Tuning the FICO Xpress Optimizer

47.01

FICO® Xpress Optimization

FICO

©2010–2026 Fair Isaac Corporation. All rights reserved. This documentation is the property of Fair Isaac Corporation ("FICO"). Receipt or possession of this documentation does not convey rights to disclose, reproduce, make derivative works, use, or allow others to use it except solely for internal evaluation purposes to determine whether to purchase a license to the software described in this documentation, or as otherwise set forth in a written software license agreement between you and FICO (or a FICO affiliate). Use of this documentation and the software described in it must conform strictly to the foregoing permitted uses, and no other use is permitted.

The information in this documentation is subject to change without notice. If you find any problems in this documentation, please report them to us in writing. Neither FICO nor its affiliates warrant that this documentation is error-free, nor are there any other warranties with respect to the documentation except as may be provided in the license agreement. FICO and its affiliates specifically disclaim any warranties, express or implied, including, but not limited to, non-infringement, merchantability and fitness for a particular purpose. Portions of this documentation and the software described in it may contain copyright of various authors and may be licensed under certain third-party licenses identified in the software, documentation, or both.

In no event shall FICO or its affiliates be liable to any person for direct, indirect, special, incidental, or consequential damages, including lost profits, arising out of the use of this documentation or the software described in it, even if FICO or its affiliates have been advised of the possibility of such damage. FICO and its affiliates have no obligation to provide maintenance, support, updates, enhancements, or modifications except as required to licensed users under a license agreement.

FICO is a registered trademark of Fair Isaac Corporation in the United States and may be a registered trademark of Fair Isaac Corporation in other countries. Other product and company names herein may be trademarks of their respective owners.

Patent(s): www.fico.com/en/patents

FICO® Xpress Optimizer 47.01 (FICO® Xpress 9.9)

Deliverable Version: A

Last Revised: 15 May, 2026

Contents

Introduction 1

Invoking the Tuner 3

Optimizer Console . . . 3
Optimizer Library C API . . . 3
Xpress Mosel . . . 3
Builder Component Library (Xpress BCL) . . . 3
Xpress Workbench . . . 4
Using a Control Setting . . . 4

Creating a Problem File 5

Optimizer Library C API . . . 5
Xpress Mosel . . . 5
Builder Component Library (Xpress BCL) . . . 6

Using the Xpress Optimizer Console 7

Manual Tuning Overview 8

Multi-threaded Solves . . . 8

Presolve 10

Presolve Controls Summary . . . 10

Solving the continuous relaxation 11

Continuous Solve Controls Summary . . . 12

Heuristics 13

Heuristic Controls Summary . . . 16

Cutting 17

Cutting Controls Summary . . . 18

Branch-and-Bound Tree Search 20

Branch-and-Bound Controls Summary . . . 22

Appendix 23

A Contacting FICO 23

FICO Customer Support 23
Documentation 23
FICO Learning 24
Sales and maintenance 24
About FICO 24

Introduction

The FICO Xpress Optimizer is a powerful optimization engine that sits at the core of the FICO Xpress Optimization Suite. It can solve a wide range of mathematical programming problems, such as linear problems, quadratic problems, and mixed integer versions of these. The Xpress Optimizer can be called either indirectly, through the Mosel modeling language, or directly through various programming APIs, including the Builder Component Library (BCL).

The FICO Xpress Optimization Suite also includes the ability to solve general nonlinear problems to proven global optimality, using the capabilities of FICO Xpress Global, or, alternatively, FICO Xpress Nonlinear to get locally optimal solutions for nonlinear problems. This guide is primarily concerned with the Xpress Optimizer. SOme suggestions apply for Xpress Global as well, and for specific information on how to get the most out of Xpress Nonlinear, please refer to the Xpress Nonlinear Reference Manual.

The Xpress Optimizer employs many diverse techniques when solving a given problem. Which technique to apply and with how much effort to put into it is highly configurable. Most of the solver components' behavior can be modified — or tuned — by setting some control parameters. The full list of these control parameters is available in the Xpress Optimizer Reference Manual. The default values for these controls have been selected to provide the best performance across a wide variety of problems. But what works best on average might not be the best possible strategy for your specific problem. If performance is critical, it is often possible to fine-tune the control parameters to improve on the default behavior for a selected set of problems.

For instances that should be tuned for heuristic performance, we recommend trying HEUREMPHASIS=1. This setting addresses instances that will likely not solve to proven optimality within a given time limit. It aims at reducing the primal-dual gap in an early stage of the phase, primarily by running more aggressive heuristics.

For further fine-tuning, the Xpress Optimizer includes a built-in Tuner accessible from the console, through an API, or within the Xpress Mosel modeling language itself. It can tune all problem types supported by both the Xpress Optimizer, Xpress Global and Xpress Nonlinear, as long as the necessary solver licenses are available. The Tuner is also available through the Workbench integrated development environment. Section "Using the Tuner" of the Xpress Optimizer Reference manual describes the Tuner in more detail.

The Tuner is a tool intended to automate the process of discovering better control parameter settings for either a single problem instance or a small collection of problem instances. When tuning a single problem instance, it works directly with the problem currently loaded into the Xpress Optimizer. It will systematically test the problem against a range of different control settings. Any improved control settings discovered will be printed in the output log. At the end of the tuning session, the overall best-performing control collection is printed and applied to the problem itself. A small summary will be included, highlighting the contribution from each individual control setting.

A single tuning run will typically involve solving each problem at least 100-200 times and can therefore become very expensive for more difficult problems. The Tuner also includes a convenient feature to automatically include permuted versions of the problem in the tuning set to ensure that the suggested control parameter settings are more robust against random fluctuations of the solution path. This will significantly increase the tuning time, though, as each additional permutation introduces an additional solve for each control combination.

This guide is intended to help you get the best performance out of the Xpress Optimizer for harder problems, in particular mixed integer problems (MIPs). For purely continuous linear and quadratic problems, the tuning choices are fairly limited, and the choice usually amounts to selecting the most appropriate basic algorithm. The number of procedures applied within a mixed integer problem solver goes far beyond that of a continuous problem solver, with an accompanying wide array of tuning options.

In the following section, we will describe how to invoke the automatic built-in Tuner through the different interfaces. The bulk of this document focuses on how to tune manually using the Xpress Optimizer console and how to use the hints provided in the output logs to select better control settings.

Invoking the Tuner

How to invoke the Tuner in the Xpress Optimizer depends on how the Optimizer is being accessed.

Optimizer Console

In the Xpress Optimizer console, the Tuner is invoked using the `TUNE` command. This will start the Tuner on the problem currently loaded into the Optimizer.

Optimizer Library C API

The Tuner can be invoked on any `XPRSprob` currently set up in memory, by calling the `XPRStune` function, which has the following declaration:

```
int XPRS_CC XPRStune(XPRSprob prob, const char* _sflags);
```

Xpress Mosel

Invoking the Tuner from within a Mosel model is as simple as adding the `XPRS_TUNE` flag to any call to `minimize` or `maximize`. For example, a MIP model with objective function *myObjective* using Barrier for the initial relaxation can be tuned by calling:

```
minimize(XPRS_TUNE+XPRS_BAR, myObjective)
```

The outcome of the tuning session will be printed in the standard output log. After tuning the model, the Optimizer will perform a final solve with the discovered control parameters fixed. Model execution can therefore continue as normal, and this way, it is possible to tune a specific solve within a model that involves multiple, iterative solves.

Builder Component Library (Xpress BCL)

The Tuner is not accessible directly from Xpress BCL. You first need to obtain a handle to the underlying Xpress Optimizer object, on which you can then invoke the Tuner, as mentioned above. This would typically look like:

```
XPRBprob bcl_prob;
XPRSprob opt_prob;
...
XPRBloadmat(bcl_prob);
opt_prob = XPRBgetXPRSprob(bcl_prob);
XPRStune(opt_prob, "");
```

Xpress Workbench

The tuner can be invoked directly from Workbench via `Show Run Dialog...` in the `Run` drop-down menu.

Using a Control Setting

Independent of how the Xpress Optimizer is accessed, it is also possible to tell it to automatically tune the next problem being solved. This is done by setting the control parameter `TUNERMODE=1`. The next time a problem is solved, whether through the console, an API, or Mosel, the Optimizer will first run a tuning session on the problem. Then the discovered control settings are applied to the problem, and a normal solve is executed.

Creating a Problem File

For manually tuning a problem, we recommend using the Optimizer console. It is flexible and it is simple to experiment with control settings here and observe the results through the output log.

Before you start experimenting with your problem, you need to save it in a file format that the Xpress Optimizer console can read. We recommend using the *MPS* file format since it preserves the ordering of constraints and variables. It is an unfortunate yet known fact that simply reordering constraints and variables in a mixed integer problem can lead to different solution paths and, thereby, vastly different solve times.

To truly reproduce a solve exactly from within your Mosel model or your application in the console, you can use the *saved state* file format, which is a binary file format. It is possible to ask the Xpress Optimizer to save its internal state to a file, from which the Optimizer console can later restore the state. This saved state file is not humanly readable and is tied to the specific platform and release version of Xpress that it was created with. Note that this saved state file also includes any changes you have made to the default control parameter settings.

How to create an MPS file depends on the type of interface you use for accessing the Xpress Optimizer:

Optimizer Library C API

Once the problem has been loaded into the Xpress Optimizer, it can be written to an MPS file using the API call *XPRSwriteprob()*:

```
XPRSwriteprob(prob, "myproblem", "");
```

This will save the current problem in `prob` in the MPS format to the file `myproblem.mps`.

In order to create the binary saved state file, you should add the call

```
XPRSsave(prob);
```

just before any call to *XPRSmipoptimize()*. The *XPRSsave()* function will write to a file whose name is given by the name of the problem itself. This name can be changed using *XPRSsetprobname()*. If the problem name is ”myproblem”, then *XPRSsave()* will create a file called `myproblem.svf`.

Equivalent functions exist for the Java, .Net and VB APIs. Please refer to the appropriate reference manual.

Xpress Mosel

In Mosel a problem will be passed to the Xpress Optimizer when you call the *minimize* or *maximize* procedures. It is always recommended to have the Xpress Optimizer write the problem it holds in memory directly to file, instead of using Mosel’s own *exportprob* procedure. To do so, you first tell Mosel to load the problem into the Xpress Optimizer, and then to write it out. Assuming your objective function in the model is called `myObjective`, you should insert the following two lines just before any call to *minimize/maximize*:

```
loadprob(myObjective)
writeprob("myproblem", "")
```

This will create a file named `myproblem.mps`.

It is also possible to create the saved state file from within Mosel; just replace the call to *writeprob* with a call to *savestate*, as in:

```
loadprob(ObjectiveFunction)
savestate("myproblem.svf")
```

Make sure to include the `.svf` extension in the name, since this is expected by the Xpress Optimizer.

Builder Component Library (Xpress BCL)

In Xpress BCL you need to obtain a handle to the underlying Xpress Optimizer object. With it you can then create the problem file using the standard Xpress Optimizer library functions (refer to the above section on using the *"Optimizer Library C API"*. You first need to tell BCL to load

```
XPRBprob bcl_prob;
XPRSprob opt_prob;
...
XPRBloadmat(bcl_prob);
opt_prob = XPRBgetXPRSprob(bcl_prob);
XPRSwriteprob(opt_prob, "myproblem", "p");
```

To create the saved state file, you simply replace the call to *XPRSwriteprob()* with a call to *XPRSsave()*.

Using the Xpress Optimizer Console

Every standard installation of the FICO Xpress Optimization Suite includes the Xpress Optimizer console application. You start it by typing the command `optimizer` at your command prompt, and you should see something like:

```
>optimizer
FICO Xpress Solver 64bit v9.0.0 Oct 17 2022
(c) Copyright Fair Isaac Corporation 1983-2026. All rights reserved
 Optimizer v41.01.01    [/C:\xpressmp\bin\xprs.dll]
[xpress]
```

For a full description of how to use the Xpress Optimizer console and the meaning of each control parameter, please refer to the Xpress Optimizer Reference Manual, which is also available online at

https://www.fico.com/fico-xpress-optimization/docs/latest

For the purpose of tuning solve performance for a problem, the only basic commands you will need to know are:

`readprob`	Read a problem stored in the MPS (or LP) file format.
`restore`	Restore a problem from a saved state file.
`lpoptimize`	Solve a purely continuous problem.
`mipoptimize`	Solve a mixed integer problem.
`tune`	Apply automatic tuning to the problem.
`dumpcontrols`	Display all the current non-default control parameter settings.
`setdefaultcontrol`	Reset a control parameter to its default setting.

Basic information about these commands and their arguments is available in the console by typing help followed by the name of the command.

To change a control in the console, you simply assign it a new value on the command line, such as

```
[xpress] PRESOLVE = 0
```

Note that control names are case insensitive. To see the current value of a control, just type the name of it:

```
[xpress] PRESOLVE
0
```

Alternatively, use the `dumpcontrols` command to see a list of all controls that you have set.

Some controls are *bit-vector* controls. These are special integer controls where each of the 32 bits making up the integer is typically used to turn a single feature on or off. The bits are numbered from 0 to 31 and the integer value corresponding to a bit k is calculated as 2^k. For example, to set bits 1, 4 and 8 of a control, you would set it to the integer value: $2^1 + 2^4 + 2^8 = 2 + 16 + 256 = 274$.

Manual Tuning Overview

Solving a mixed integer problem usually involves five stages, depending on how hard the problem is:

1. Presolve.
2. Solving the initial LP relaxation.
3. Heuristics.
4. Cutting.
5. Branch-and-bound tree search.

This guide addresses those five parts of the solving process separately in the following sections.

How to tune the Xpress Optimizer for a particular problem depends very much on what your goal is:

Find any solution.	For some problems it can be hard to find that first integer solution. The emphasis in such a case will often be on heuristics and on guiding the branch-and-bound dives away from potential conflicts.
Find good solutions quickly.	If the goal is to find better solutions quicker, without necessarily providing any bound on how good the solutions are, the emphasis will usually be on heuristics and a fast branch-and-bound search of the solution space. Note that the Optimizer has a special mode with a focus on primal heuristics. To activate this mode, set HEUREMPHASIS=1. We advise trying this first before conducting a long tuner run or trying various heuristic controls individually.
Solve the problem to optimality.	The Optimizer is by default tuned towards finding a proven optimal solution to a problem. Default settings balance heuristics for finding good solutions with the necessary raising of the best bound to prove optimality.

You can customize the Tuner for your individual goals. The `TUNERMETHOD` control allows using different ground sets of controls that the Tuner evaluates. The `TUNERTARGET` control allows setting different measures by which individual solves should be compared, including the time, the gap, the final solution value.

Multi-threaded Solves

The Xpress Optimizer is a multi-threaded solver and will always try to use all available processing cores in your computer. It automatically detects the number of cores available and will limit the number of parallel threads it can start accordingly.

If your Xpress Optimizer has to share system resources with other applications that might be running simultaneously, it might be necessary to limit how many threads, and thereby how many cores, the

Xpress Optimizer is allowed to use. Use the THREADS control to set such a limit. By setting THREADS=1, you force the Xpress Optimizer to use a single thread only, which effectively turns off all parallelization. Note that in the case of Tuner runs, THREADS controls the number of threads per solve run, while the additional TUNERTHREADS control determines how many different settings are tried in parallel solver runs. Consequently, the product of THREADS and TUNERTHREADS should be equal to or less than the number of available threads on the machine.

Presolve

The presolver (also called preprocessor) is used to reduce the problem's initial size by dropping unnecessary constraints and variables. It will also attempt to tighten the problem, *i.e.*, modify bounds and constraints. This leads to non-optimal or non-integer solutions being removed while preserving at least one optimal solution.

Presolving is often the most crucial part when solving a problem, and the reduction in size resulting from presolve can cause a significant difference in solution time. Presolving is always the first step when solving a problem, and the output will typically look something like:

```
[xpress] mipoptimize
Minimizing MILP air05 with default controls
Original problem has:
       427 rows        7195 cols     59316 elements     7195 entities
Presolved problem has:
       335 rows        6133 cols     36203 elements     6133 entities
LP relaxation tightened
```

If the displayed problem size is significantly smaller than that of the initial problem, you know that presolve is working well. It is possible to turn off presolving by setting `PRESOLVE=0`, but it is rarely beneficial to do so.

If a problem contains binary variables, the presolver will by default perform probing on them. Probing involves fixing each binary variable in turn to 0 and 1 and then analyzing any implications. This procedure often leads to the binary itself being fixed due to conflicts or to other variables being dropped due to them being directly implied by the binary. It is possible to set the level of probing the Optimizer performs using the `PREPROBING` control. There are four possible values, ranging from `0` (no probing) to `3` (full probing). If the presolve is taking too much time, it is often probing that is the cause. Otherwise, try setting `PREPROBING=3` (or `2` if `3` takes too much time) and watch how it affects the size of the presolved problem.

It is usually only when solving many very easy MIPs that presolve time can become an issue. If so, some time can be saved by restricting it to performing continuous reductions only. In other words, prevent it from performing any reductions that depend on the integrality of variables. This is done by turning on bit 9 of `PRESOLVEOPS`, *i.e.*, by setting `PRESOLVEOPS` = 511 + 2^9 = 1023, where `511` is the default value. Allowing the Optimizer to perform continuous only reductions can also be useful if you wish to provide a starting basis for a linear MIP. By preventing integrality reductions, you ensure that the validity of the basis is preserved.

Presolve Controls Summary

`PRESOLVE`	Turn presolving on (1) or off (0).
`PRESOLVEOPS`	An integer bit-vector control, where each bit turns an individual presolving feature on or off.
`PREPROBING`	Controls the level of presolve probing on binary variables (0-3).

Solving the continuous relaxation

Getting that first feasible solution to the continuous relaxation of a MIP can often be quite hard. It is always worth trying the three main LP algorithms available in the Xpress Optimizer: *primal simplex, dual simplex, hybrid gradient* and *barrier*. These LP algorithms are not selected through a control but are instead specified as options to the `lpoptimize` and `mipoptimize` commands. Use one of the following options to force a particular algorithm selection:

`>mipoptimize –p` Primal simplex

`>mipoptimize –d` Dual simplex

`>mipoptimize –b` Barrier or Hybrid gradient (if `BARALG` is set to 4)

Note that primal simplex, the dual simplex and the hybrid gradient methods cannot be applied to a continuous problem with quadratic or conic constraints. The Optimizer will always solve such problems with the Barrier algorithm, irrespective of any control specifications.

The default choice of algorithm for a linear MIP depends on the number of *threads*. *Dual simplex* is always the default choice for a single-threaded solve. When more threads can be started, the Xpress Optimizer will run a *concurrent LP* solve, where it starts *primal simplex*, *dual simplex* and *barrier (or hybrid gradient)* in parallel. It stops with the first one to solve the continuous relaxation. There is some overhead in running all of the algorithms in parallel. Thus, if one particular algorithm always comes up as the winner, it is better to force that one to always be run on its own. Besides the options above, a particular algorithm can also be forced by setting `LPFLAGS` to 1 (for dual), 2 (for primal), or 4 (for barrier). If the overhead is mostly in synchronization, you can try to set `DETERMINISTIC` to 2, as an intermediate way of avoiding a long waiting time without enforcing a certain algorithm but keeping the main MIP search deterministic.

On some MIPs, the choice of the initial algorithm has an effect beyond the time to solve the initial continuous relaxation. If the relaxation has many optimal solutions (a property known as dual degeneracy), the choice of algorithm will also affect which of these optional solutions is found. In general, primal simplex will find a least fractional solution while barrier and hybrid gradient tend to find solutions with more fractionalities. This can substantially affect the subsequent cutting and heuristics.

For dual simplex the main controls to consider are `DUALGRADIENT` and `DUALPERTURB`. The `DUALGRADIENT` control specifies how dual simplex selects its pivots and can be a value from 0 to 3, with -1 leaving the choice to the Optimizer. Dual simplex uses cost perturbation to get out of a ”sticky” solution. The `DUALPERTURB` control is used to specify the initial random perturbation to apply to the objective function, which can be useful for *dual degenerate* problems. Note that since dual simplex is also the default algorithm for reoptimizing the continuous relaxation of any sub-problem while solving a linear MIP, setting `DUALPERTURB` will affect the whole MIP solve. `DUALSTRATEGY` is relevant for numerically challenging problems, since it determines how cycling is handled.

With primal simplex, the main control is `PRICINGALG`. The values to try here are from -1 to 3.

Continuous Solve Controls Summary

`-p, -d, -b` options	Options for `lpoptimize`/`mipoptimize` to select the algorithm when solving the initial continuous relaxation: *primal simplex*, *dual simplex* or *barrier (or hybrid gradient)*.
`BARALG`	Set it to 4 to use the hybrid gradient method instead of the Newton barrier algorithm.
`CONCURRENTTHREADS`	Number of threads to use for a concurrent LP solve.
`DUALGRADIENT`	Dual simplex pivot selection strategy (0 3).
`PRICINGALG`	Primal simplex pivot selection strategy (-1 3)
`DUALPERTURB`	Initial perturbation value for each dual solve (try 0 for none, or *e.g.* `1e-4`).

Heuristics

The Xpress Optimizer contains a wide variety of heuristics to help find feasible solutions during a MIP solve. They generally fall into one of the following three categories:

Simple rounding heuristics	Very fast heuristics that appliy some simple rules to the solution of the continuous relaxation in order to produce a feasible MIP solution. These are typically run on every node. Specific controls: *none*
Diving heuristics	Start from the continuous relaxation solution to a node and combine rounding and fixing of MIP entities with occasional reoptimization of the continuous relaxation to construct a better quality MIP solution. They are run frequently on both the root node and during the branch and bound tree search. Specific controls: `HEURFREQ`, `HEURDIVESTRATEGY`, `HEURDIVESPEEDUP`.
Local search heuristics	The local search heuristics are generally the most expensive heuristics and involve solving one or more smaller MIPs whose feasible regions describe a neighborhood around a candidate MIP solution. These heuristics are run during the root cut-loop and typically on every 500 - 1000 nodes during the tree search. Specific controls: `HEURSEARCHFREQ`, `HEURSEARCHEFFORT`, `HEURSEARCHROOTSELECT`, `HEURSEARCHTREESELECT`, `HEURSEARCHROOTCUTFREQ`.

In addition to these heuristics, the Optimizer also contains a *feasibility pump* heuristic. It is off by default, and you should only consider it when all other heuristics have failed to find a solution. It can be turned on by setting the `FEASIBILITYPUMP` control to either 1 (run always) or 2 (run when no MIP solution exists).

For problems with very hard continuous relaxation, the Optimizer features the Pre-root parallel heuristic phase, which is disabled by default, but is enabled either through `HEUREMPHASIS=2` or through the controls `PREROOTEFFORT` or `PREROOTWORKLIMIT`, respectively. If enabled, this phase runs a diverse strategy of combinatorial and sub-MIP heuristics in parallel on all available threads. The main work limit for this phase can be directly set via `PREROOTWORKLIMIT`. Using the `PREROOTEFFORT` control lets the Optimizer decide on a suitable work limit depending on the problem characteristics. Here, it is advisable to start with `PREROOTEFFORT=1`. In order to emphasize Pre-root parallel heuristics further or reduce the effort, set this control value to a value that is larger than 1/smaller than 1, respectively. In both cases the control value acts as a factor for the work limit decided with `PREROOTEFFORT=1`. The `PREROOTTHREADS` control can be used to specify a number of parallel threads. The default value uses all available threads, as decided by the `THREADS` control. The strategy in pre-root parallel heuristic phase benefits from more available threads. However, it might be necessary to tune down the number of threads used if the memory footprint of this phase becomes too large.

The set of heuristics that the Optimizer should run can be set indirectly using the `HEUREMPHASIS` control. The default is that all heuristics run according to their individual controls. The interpretations of the settings of `HEUREMPHASIS` are roughly:

-1 – Default heuristics.
0 – Disable all heuristics.
1 – Focus on reducing the primal-dual gap in the early part of the search. This addresses problems where the goal is to achieve a small gap but not neccessarily solving them to optimality.
2 – Extremely aggressive search heuristics, including pre-root parallel heuristics. Good for some problems.

In the output log, single letters at the start of the line indicate a heuristic solution, such as in this example:

```
Starting root cutting & heuristics
 Its Type    BestSoln    BestBound   Sols    Add    Del     Gap      GInf   Time
k           16.250000    11.204731      1                  31.05%       0      0
   1  K     16.250000    12.435740      1     58      0    23.47%      21      0
   2  K     16.250000    12.971545      1     71     94    20.18%      24      0
d           16.000000    12.971545      2                  18.93%       0      0
r           15.250000    12.971545      3                  14.94%       0      0
Heuristic search started
R           14.750000    12.971545      4                  12.06%       0      0
Heuristic search stopped
```

Here, the first solution with objective value 16.25 was found by rounding heuristic strategy 'k' before the first round of cutting.

The letters at the start of the line identify the heuristic responsible for finding the solution. The following table helps map letters to heuristics and the primary control for enabling or disabling the specific heuristic.

Letter	Heuristic	Controls
*	Integral LP relaxation (B&B node solution)	
E	Solution found during calculation of branching estimates	
U	User provided solution	`USERSOLHEURISTIC`
a-z	Diving heuristics	`HEURDIVESTRATEGY`
R	Relaxation Induced Neighbourhood Search (RINS)	`HEURSEARCHROOTSELECT`, `HEURSEARCHTREESELECT`
L	LP centered local search	`HEURSEARCHROOTSELECT`, `HEURSEARCHTREESELECT`
M	MIP centered local search	`HEURSEARCHROOTSELECT`, `HEURSEARCHTREESELECT`
C	MIP solution combination local search	`HEURSEARCHROOTSELECT`, `HEURSEARCHTREESELECT`
Z	Objective-free local search	`HEURFORCESPECIALOBJ`
A	Analytic Center local search	`HEURFORCESPECIALOBJ`
F	Feasibility pump	`FEASIBILITYPUMP`
S	Set packing/partitioning/covering heuristics	
B	Heuristics branching on constraints	
T	Trivial heuristic	*Always enabled (unless all off)*
P	Solution found by parallel background heuristics	
H	Shift-and-propagate	*HEURSHIFTPROP*
N	Local solver heuristic (Xpress Global only)	*GLOBALLSHEURSTRATEGY*

The *diving* heuristics are typically run between each round of cuts on the root problem and then once for every k nodes solved, where the frequency k is determined by the HEURFREQ control. A similar control, HEURSEARCHFREQ, exists for local search heuristics.

There are 18 preset diving heuristic strategies. The Xpress Optimizer will typically test 4-6 of them before and after the root problem cutting and pick the most promising strategy for the tree solve. It is possible to override this selection by setting HEURDIVESTRATEGY to a value from 1 to 18. Testing all diving strategies is part of the TUNERTARGET=8 setting.

The *local search* heuristics rely on creating and solving a reduced MIP and can therefore be quite expensive to run. Normally, only two local search heuristics will run on the root problem, at most every five cut rounds. During the branch-and-bound search, the local search heuristics will be run once for every k nodes, where the frequency k is determined by the HEURSEARCHFREQ control.

The Xpress Optimizer has several local search heuristics, but only some of them are on by default. The additional heuristics can be turned on using the HEURSEARCHROOTSELECT control (for the root problem) or the HEURSEARCHTREESELECT control (for the branch-and-bound search). These are both bit-vector controls where each individual bit is used to turn a heuristic on or off. The values of these bits are:

Bit 0	Local search with a large neighborhood. The default heuristic.
Bit 1	Local search with a small neighborhood, centered around the continuous relaxation solution.
Bit 2	Local search with a small neighborhood, centered around the integer solution. Default on for the root node only.
Bit 3	Local search with a neighborhood given by multiple integer solutions.
Bit 4	*[unused]*
Bit 5	Local search without an objective function. Typically expensive, but can be useful for finding a first solution. Applicable to the root only.
Bit 6	Local search with an auxiliary objective function. Typically expensive, but can be useful for finding a first solution. Applicable to the root only.

To test whether running any of these non-default heuristics is useful, it is best to turn all of them on and observe whether additional solutions are discovered. To do so, enable all the bits of HEURSEARCHROOTSELECT, by setting it to 127 (or equivalently -1).

When the emphasis is on **finding high-quality solutions quickly**, we recommend setting HEUREMPHASIS to 1 or even 2. Alternatively, you could try to find a specific combination of HEURSEARCHROOTCUTFREQ, HEURSEARCHFREQ, HEURSEARCHROOTSELECT and HEURSEARCHTREESELECT that works best for your problem.

Running heuristics is particularly useful when the branch-and-bound tree search has to dive to considerable depths. Observe the ”Depth” column of the following log segments:

```
Node     BestSoln    BestBound   Sols Active  Depth      Gap      GInf    Time
   1  1.32251E+11  1.32210E+11      3      2      1     0.03%     1897     348
...
  10  1.32251E+11  1.32210E+11      3      9      4     0.03%     1937     360
...
 100  1.32251E+11  1.32210E+11      3     11     93     0.03%     1827     442
...
1000  1.32251E+11  1.32210E+11      3     11    992     0.03%     1076     783
...
2000  1.32251E+11  1.32210E+11      3     11   1992     0.03%      581    1087
...
3000  1.32251E+11  1.32210E+11      3     11   2992     0.03%      408    1469
```

Here the ”Depth” column keeps increasing, while the ”GInf” column (number of fractional variables) steadily decreases. This is a sure indicator that the branch-and-bound solve is still on its first dive towards integer feasibility. Without heuristics, it will not find a new solution before the ”GInf” column reaches zero. Heuristics can be used to find improved solutions at any point during this dive.

On the other hand, if the emphasis is on **solving to optimality**, heuristics have been found to have only a little effect on the overall solve time, except in special circumstances.

Finally, it is possible to increase the amount of effort put into each run of the local search heuristics through the `HEURSEARCHEFFORT` control. The default value is 1.0, and increasing it to 2.0 is the equivalent of asking the heuristic to try twice as hard.

Heuristic Controls Summary

`HEUREMPHASIS`	Select how much emphasis to put on primal heuristics (0 off to 3 aggressive).
`HEURDIVESTRATEGY`	Select one of the preset 10 diving heuristic strategies (0 off, or 1 10).
`HEURFREQ`	Run a diving heuristic for every `HEURFREQ` nodes solved.
`HEURSEARCHROOTSELECT/HEURSEARCHTREESELECT`	Bit-vector control to enable or disable specific local search heuristics, on the root problem or during the branch-and-bound search, respectively.
`HEURSEARCHROOTCUTFREQ`	How frequently to run local search heuristics between rounds of root cutting.
`HEURSEARCHFREQ`	Run a local search heuristic for every `HEURSEARCHFREQ` nodes solved.
`HEURSEARCHEFFORT`	Multiplier on the effort spent on local search heuristics.

Cutting

There are several classes of cutting plane algorithms available for the Optimizer to tighten a MIP formulation. Cutting planes (or just cuts) will, in general, be created and added to the formulation in several rounds on the root problem and occasionally during the branch-and-bound search.

The naming of the controls governing cutting reflects their historical legacy and can be slightly misleading. What is referred to in the Xpress Optimizer as *cover cuts* started out as knapsack cover cuts but will today include several other classes of cuts, such as *mixed integer rounding (MIR) cuts, clique cuts, flow-path cuts, multi-commodity flow (MCF) cuts, zero-half cuts*, and more. These are all cuts derived directly from the problem constraints, sometimes referred to as combinatorial cuts. The related controls are called `COVERCUTS` and `TREECOVERCUTS`. These controls set an upper limit on the number of rounds of cuts to apply, on the root node or an in-tree node, respectively. The default settings of -1 will usually allow for roughly 20 rounds on the root node and just one round on an in-tree node.

The other main group of cuts is derived from rows of the simplex tableau of the optimal linear relaxation solution (they are not available for quadratic problems). These are *mixed integer Gomory cuts*, or *Lift-and-Project cuts*. The corresponding controls to limit the number of rounds for these cuts are `GOMORYCUTS` and `TREEGOMORYCUTS`.

For cutting in the tree, another important control is `CUTFREQ`. It sets the frequency by which to apply cuts during the branch-and-bound tree search. When set to a value `k`, cutting will be applied to every `k`'th level of the tree. Leaving it at -1 will let the Optimizer decide whether to apply cuts at any given node. This default strategy tends to create more frequent cutting near the top of the tree. Note that a setting of 0 will completely disable cutting during the tree search. You can use the `CUTDEPTH` control to limit the maximum tree depth at which the Optimizer will apply cutting. A setting of `k` will disable cutting for any node deeper than level `k` in the tree.

Another crucial aspect of cutting is the choice of how many cuts to add to the node problem. The more cuts, the harder it becomes to resolve the continuous relaxation. There is a tradeoff between the additional effort in solving the relaxations and the strengthening provided by cuts. The `CUTSTRATEGY` control provides three pre-set levels to determine how many cuts to add to the problem:

`-1`	Automatic.
`0`	Turn cuts off completely.
`+1`	Low level of cuts.
`+2`	Medium level of cuts.
`+3`	High level of cuts.

The output log from a solve will usually provide some hints on whether it is beneficial to increase the `CUTSTRATEGY` control. Consider the following example output from a solve:

```
 Its Type    BestSoln    BestBound   Sols    Add    Del     Gap      GInf   Time
a        -4250519213. -983637913.7      1                76.86%        0      0
d        -4186315351. -983637913.7      2                76.50%        0      0
   1  K  -4186315351. -2301857560.      2    734      0   45.01%      122      1
   2  K  -4186315351. -2444516200.      2    510    173   41.61%      103      1
```

```
   3  K  -4186315351. -2479120937.      2    447    353   40.78%       98      1
   4  K  -4186315351. -2486387651.      2    389    302   40.61%       91      1
...
  16  K  -4186315351. -2568819556.      2    295    286   38.64%       71      3
  17  K  -4186315351. -2570860713.      2    290    284   38.59%       74      4
  18  K  -4186315351. -2571188037.      2    281    266   38.58%       71      4
  19  K  -4186315351. -2575283602.      2    276    269   38.48%       61      4
  20  K  -4186315351. -2587960341.      2    277    511   38.18%       77      4
```

Here one can observe that the Optimizer keeps adding a substantial number of cuts in each round. This is a clear indicator that it has identified many more violated cuts than it is allowed to add by the default `CUTSTRATEGY` setting. The first thing to try on such a problem is to set `CUTSTRATEGY = 3`. Indeed for this problem, the solve is clearly improved:

```
 Its Type    BestSoln    BestBound   Sols    Add    Del     Gap      GInf   Time
a        -4250519213. -983637913.7      1                  76.86%       0      1
d        -4186315351. -983637913.7      2                  76.50%       0      1
   1  K  -4186315351. -2500370863.      2   1574      0   40.27%      122      1
   2  K  -4186315351. -2791609815.      2   2207    730   33.32%      104      2
   3  K  -4186315351. -3016999540.      2   3113   1755   27.93%       92      3
   4  K  -4186315351. -3082444620.      2   3969   2889   26.37%      107      4
...
d        -3746982101. -3413070439.      8                   8.91%       0     24
  16  K  -3746982101. -3421885848.      8   4245   4215    8.68%      104     24
  17  K  -3746982101. -3429939240.      8   4243   4240    8.46%      100     26
  18  K  -3746982101. -3437132243.      8   4234   4219    8.27%      105     28
  19  K  -3746982101. -3441666577.      8   4204   4160    8.15%      108     31
d        -3744370125. -3441666577.      9                   8.08%       0     34
  20  K  -3744370125. -3445582328.      9   4193   7561    7.98%      104     34
```

Here, the strengthening provided by the additional cuts has clearly improved the best bound. Another positive side effect is that this has helped the heuristics to find even better solutions. So overall the MIP gap improved from 38% to 8%. There is, of course, a cost to this, which is the increase in cutting time from 4 seconds to 34 seconds. But this is a fairly large problem for which the branch-and-bound search will easily take much more than 30 seconds, so the improvement is well worth it.

If the emphasis is on **finding a good solution quickly**, the cutting part can often be too expensive, and it frequently happens that the best bound is improved only marginally after the first few rounds of cuts. Once could conclude that it should be safe to reduce the number of rounds by setting `COVERCUTS` to a smaller number. However, there is a close interplay between cut rounds and local search heuristics. More rounds of root node cuts typically mean more local search heuristic calls, with different reference solutions. Thus, while increasinf or decreasing the number of cut rounds often has a huge impact on the speed at which solutions are found, it is not clear a priori in which direction the impact goes. Finally, it might sometimes be beneficial to reduce the amount of in-tree cutting through the `CUTFREQ` control, to speed up the number of nodes solved.

On the other hand, when the emphasis is on **solving to optimality**, cutting is often crucial in order to improve the best bound. Here on can try setting `CUTSTRATEGY = 3`. If cutting is useful on the root it is often beneficial in the tree as well. Try *e.g.* setting `CUTFREQ = 4`, to cut fairly frequently.

Cutting Controls Summary

`CUTSTRATEGY`	Amount of cuts to add to the problem, expressed as one of a preset number of levels (-1 3).
`CUTFREQ`	Frequency of cutting during the tree search.
`CUTDEPTH`	Maximum depth of a node in the tree on which cutting can be applied.
`COVERCUTS, TREECOVERCUTS`	Maximum number of rounds of structural cuts to apply on the root node and on in-tree nodes, respectively.

`GOMCUTS, TREEGOMCUTS`	Maximum number of rounds of Gomory/Lift-and-Project cuts to apply on the root node and on in-tree nodes, respectively.
`CUTSELECT, TREECUTSELECT, MCFCUTSTRATEGY`	Provide detailed control of the structural cuts created during the MIP solve.

Branch-and-Bound Tree Search

For many MIPs, only running heuristics and applying cuts is insufficient to solve the problem to proven optimality. Therefore, the Xpress Optimizer will have to enter the branch-and-bound search. This process recursively selects a single undecided variable and splits the current problem into two easier subproblems by forcing the variable either up or down.

The most important aspect of this search is the selection of the *branching variable*. Good choices here can lead to solution times orders of magnitude faster than bad choices. If the focus is on **finding good solutions quickly**, then the order in which the two subproblems are searched is also important.

The Xpress Optimizer uses a wide variety of information in order to guide the choice of a branching variable. This includes the outcomes of past branching choices, some strong branching, and some quick estimates based on the local subproblem.

By default, the Optimizer will use *pseudo costs* as its main criteria for selecting the branch entity. This is an estimated cost of moving a variable either up or down by a unit amount. This cost is multiplied by the distance a variable has to move to reach the next integer value, which provides an estimated change in the objective function. Strong branching is used to initialize these pseudo costs.

The basic procedure for selecting a branching variable, and the controls that affect each step are as follows:

1. Pre-filter the set of candidate variables using very cheap estimates.
   `SBSELECT`: determine filter size
2. Evaluate and rank selected candidates.
   `SBESTIMATE`: local ranking function
3. Initialize pseudo costs using strong branching
   `SBBEST`: Number of variables to strong branch on.
   `SBITERLIMIT`: LP iteration limit for strong branching.
4. Select the best candidates using a combination of pseudo costs and the local ranking function.

The overall amount of effort put into this process can be adjusted using the `SBEFFORT` control. The default value is 1.0, and a value of 2.0 tells the Xpress Optimizer to put in twice the default effort.

Strong branching can be turned off completely by setting either `SBBEST=0` or `SBITERLIMIT=0`. This also disables the use of pseudo costs, which causes step 3 above to be skipped. In that case, only the local ranking function determined by `SBESTIMATE` will be used.

There are six choices for the `SBESTIMATE` local ranking function:

1	Transformed distance measure (expensive)
2	Simplex tableau based (expensive)
3	Use probing/logical implications to create cost estimates.
4	Count probing/logical implications.

5 Simple cost estimates (quick)

6 Row influence measure (quick)

If pseudo costs are not working well, it is recommended to turn off strong branching (set SBBEST=0) and try at least SBESTIMATE = 1, 2 or 3. It is usually sufficient to let the Xpress Optimizer complete a couple of dives in order to get a good idea of which ranking function works best. Look *e.g.* for the function that raises the Best Bound the most after a fixed number of nodes.

If the focus is on **finding good solutions quickly**, then setting SBBEST=0 and SBESTIMATE=5 or 6 will provide the quickest dives.

The other aspect of the branch-and-bound search is how to select the next node to work with. This choice can have a considerable impact on the time when improving solutions are found. The time to proven optimality, however, is often not affected much by node selection. The Xpress Optimizer performs the search in dives, whereby it picks a single active node from the node pool and then performs a dive on it. In a dive it creates two new subproblems by branching on a variable, picks one of the subproblems and repeats by branching on another variable. This process continues until it reaches a subproblem without any fractional variables to branch on, or for which no improved solutions exist. The selection of an active node is also referred to as a global backtrack. During a dive, the Xpress Optimizer might also perform local backtracks, where it switches between sibling nodes.

The main controls for selecting the next active node to work on is BACKTRACK. The default value of 3 (best bound) will work best on almost all problems. The related control BACKTRACKTIE will be used to break ties if two nodes have the same ranking according to BACKTRACK. Possible values range from 2 to 12 and it is mostly a matter of experimentation to find which works best.

The control BRANCHCHOICE can be used to select which of the two new child subproblems is explored first.

Finally, it is possible to use *directives* to directly guide the Xpress Optimizer when selecting a branching variable or when deciding on the branch to follow during a dive. It is possibly to give each variable a priority value such that a variable with the lowest priority value is always branched on before one with a higher value. This can be very useful if you already know what the most important decision variables in your problem are. Please refer to the *Xpress Optimizer Reference Manual* for more information about the use of directives.

Another aspect of the tree search are restarts. These can already be triggered during the initial root cutting but may also happen in the tree. Restarts are triggered when the solver does not make enough progress or new solutions promise a faster solve when performing a full root presolve again. In this case, the current tree will be reset to the root, a full presolve will be performed again, and some internal settings may change:

```
    1945  56503433.22  55132652.45     10   1550     80    2.43%      31    190
    2049  56503433.22  55132652.45     10   1724     51    2.43%     321    193
Resetting tree to root.

Performing root presolve...

Reduced problem has:    3503 rows   15977 columns    118643 elements
Presolve dropped   :       0 rows       0 columns       185 elements
Will try to keep branch and bound tree memory usage below 9.0GB
Starting concurrent solve with dual, primal and barrier (6 threads)

                          Concurrent-Solve, 195s
...

Starting root cutting & heuristics

 Its Type    BestSoln    BestBound   Sols    Add    Del     Gap     GInf   Time
R         56488350.97  55132652.45     11                 2.40%       0    203
...
M         56107415.55  55132652.45     19                 1.74%       0    224
```

```
Cuts in the matrix         : 25
Cut elements in the matrix : 75552

Starting tree search.
Deterministic mode with up to 8 running threads and up to 16 tasks.
Heap usage: 39MB (peak 517MB, 1295KB system)
Heap usage: 39MB (peak 517MB, 1295KB system)
B&B tree size: 100k total

    Node     BestSoln    BestBound   Sols Active  Depth     Gap     GInf   Time
    2084  56107415.55  55132652.45     19      2      1    1.74%     682    231
```

In this case, as can be observed from the log, the restart managed to drop more non-zero elements. More importantly, the additional cutting rounds reduced the gap significantly. On the other hand, if this was not the case, restarting can be expensive since the previous progress is mostly thrown away, so if the restart did not manage to reduce the problem further or accelerate the progress of the tree search, it might be worthwhile to try disabling restarts by setting `MIPRESTART=0`.

Branch-and-Bound Controls Summary

`SBBEST, SBITERLIMIT`	Limit on number of strong branches and strong branch LP iterations.
`SBESTIMATE`	Local ranking function for selecting branching variables.
`SBEFFORT`	Multiplier on the effort spent on selecting a branching variable.
`BACKTRACK, BACKTRACKTIE`	How to select the next active node to perform a dive on.
`BRANCHCHOICE`	How to select the child problem to continue a dive on.
`MIPRESTART`	Whether restarts are allowed.

Appendix A

Contacting FICO

FICO provides clients with support and services for all our products.

FICO Customer Support

FICO Customer Support offers technical support and services ranging from self-help tools to direct assistance with a FICO technical support engineer. Support is available to all clients who have an active maintenance contract.

The FICO Customer Self-Service Portal (support.fico.com) is a secure web portal that allows users to open, review, and update their support cases; manage their organization's portal users; find solutions to common problems in the FICO Knowledge Base; and view the availability of their cloud applications 24 hours a day, 7 days a week.

You can find support contact information and a link to the FICO Customer Self-Service Portal (online support) on the Product Support home page (www.fico.com/en/product-support).

Please include *'Xpress'* in the subject line of your support queries.

Documentation

FICO continually looks for new ways to improve and enhance the value of the products and services we provide.

If you have comments or suggestions regarding how we can improve this documentation, let us know by sending your suggestions to techpubs@fico.com. Please include your contact information (name, company, email address, and optionally, your phone number) so we may reach you if we have questions.

FICO Learning

FICO Learning is the principal provider of product training for our clients and partners. FICO Learning offers instructor-led classroom courses, web-based training, seminars, and training tools for both new user enablement and ongoing performance support.

For additional information, visit the FICO Learning home page at www.fico.com/en/product-training or email producteducation@fico.com.

Sales and maintenance

If you need information on other Xpress Optimization products, or you need to discuss maintenance contracts or other sales-related items, contact FICO by:

- Phone: +1 (408) 535-1500 or +44 207 940 8718
- Web: www.fico.com/optimization and use the available contact forms

About FICO

FICO (NYSE:FICO) is a leading analytics software company, helping businesses in 90+ countries make better decisions that drive higher levels of growth, profitability, and customer satisfaction. Learn more at www.fico.com or contact us at www.fico.com/en/contact-us.