DYLP: a dynamic LP code

SFU-CMPT TR 2005-18

Lou Hafer

December, 2005

Abstract

DYLP is a full implementation of the dynamic simplex algorithm for linear programming. Dynamic simplex attempts to maintain a reduced active constraint system by regularly purging loose constraints and variables with unfavourable reduced costs, and adding violated constraints and variables with favourable reduced costs. In abstract, the code alternates between primal and dual simplex algorithms, using dual simplex to reoptimise after updating the constraint set and primal simplex to reoptimise after updating the variable set.

1 Introduction

DYLP is a linear programming (LP) code designed to be used as the underlying LP code in a branch-and-cut integer linear programming (IP) code. It emphasises convenience of use by the client, particularly with respect to fixing variables and adding and deleting constraints and variables. The target user population is IP algorithm developers; as such, DYLP emphasises controllability and convenience over efficiency and is capable of producing copious amounts of output for use in debugging.

DYLP implements a dynamic simplex algorithm along the lines set out by Padberg in [9, §6.6]. The core idea is that, at any given time, many of the constraints of a LP problem are loose, and many nonbasic variables are unlikely to ever be considered for pivoting because their reduced costs are very unfavourable. A rough outline of the algorithm, neglecting unboundedness, infeasibility, and implementation issues, is as follows.

From the problem supplied by the client, DYLP chooses an initial subset of constraints and variables to become the active system. This system is solved to optimality with primal simplex. DYLP then enters a minor loop where it deactivates variables whose reduced costs are worse than a threshold, activates variables whose reduced costs are favourable, and reoptimises the system with primal simplex. This minor loop is repeated until there are no more variables suitable for entry. Next, DYLP deactivates any loose constraints, activates any constraints which are violated at the current basic solution, and reoptimises with dual simplex. On regaining feasibility, it returns to primal simplex and the deactivate/activate variable loop. When there are no variables with favourable reduced costs among the inactive variables and no violated constraints among the inactive constraints, the solution is optimal.

The primal simplex algorithm used by DYLP is a two-phase algorithm. Phase I uses a dynamically modified objective to attain a primal feasible solution. Both phase I and phase II use a projected steepest edge (PSE) pricing algorithm outlined by Forrest & Goldfarb [3, algorithm 'dynamic']. There are two antidegeneracy methods. The first, referred to as 'anti-degeneracy lite', attempts to resolve ties among degenerate pivots by choosing the pivot in such a way as to make tight a hyperplane which has a desirable alignment. The second, applied when the first takes too long to resolve the degeneracy, is a perturbation algorithm which builds on a method described by Ryan & Osborne [10].

The dual simplex algorithm provides only a second phase with dual steepest edge (DSE) pricing [3, algorithm 'steepest 1'], standard or generalised pivoting, and an implementation of anti-degeneracy lite in the dual space. In the context of DYLP, it is the subordinate simplex, used for reoptimisation after adding constraints and as the initial simplex when the problem is dual feasible but not primal feasible.

The active and inactive constraint systems are maintained with the CONSYS subroutine library [5]. Basis factoring and pivoting are handled using the basis maintenance package from GLPK [6, 7].

DYLP is written in C and provides a native C interface. It can be used as a standalone simplex LP code with only a minimal shell required to generate the constraint system.

In the context of a branch-and-cut code, DYLP expects that the dominant mode of use will be successive calls to reoptimise a constraint system that is incrementally modified between calls. On request, it will maintain its internal state (constraint system, basis inverse, and support data structures) between calls to support efficient hot starts for reoptimisation. DYLP provides interface routines to support two queries commonly required in a branch-and-cut context, pricing a new variable and pricing a dual pivot. Because DYLP maintains this internal state, it does not provide a native capability to interleave optimisation and reoptimisation of distinct constraint systems.

DYLP can be used with COIN-OR [2] software through the C++ OsiDylp OSI interface class. An OSI interface object maintains a copy of the constraint system as well as providing an interface to the underlying solver. Multiple OsiDylp objects with distinct constraint systems can exist simultaneously and calls to optimise and reoptimise the systems can be interleaved. There is some loss of efficiency as the state of the underlying solver is changed, but the necessary bookkeeping is handled by the OsiDylp objects.

The next section specifies the notation used for the primal and dual problems in the remainder of the report. Sections 3 through 10 describe individual components of the implementation. Sections 11 through 15 describe the simplex algorithms and the variable and constraint management algorithms used in DYLP. Sections 16 through 18 describe the interface and parameters provided by DYLP.

2 Notation

DYLP works naturally with the minimisation problem

$$\begin{aligned} \min\ & cx \\ & Ax \leq b \\ l \leq\ & x \leq u \end{aligned} \tag{1}$$

Add slack variables s and partition $\begin{bmatrix} A & I \end{bmatrix}$ into basic and nonbasic portions as

$$\begin{bmatrix} B & N \end{bmatrix} = \left[\begin{array}{cc|cc} B^t & 0 & N^t & I^t \\ B^l & I^l & N^l & 0 \end{array}\right]$$

with corresponding partitions $\begin{bmatrix} x^B & s^B & x^N & s^N \end{bmatrix}^\top$ for x, s, and $\begin{bmatrix} b^t & b^l \end{bmatrix}^\top$ for b. The objective c is augmented with 0's in the columns corresponding to the slack variables, and partitioned as $\begin{bmatrix} c^B & 0 & c^N & 0 \end{bmatrix}$. The basis inverse will be

$$B^{-1} = \begin{bmatrix} (B^t)^{-1} & 0 \\ -B^l(B^t)^{-1} & I^l \end{bmatrix}.$$

We then have

$$\begin{aligned} \begin{bmatrix} x^B \\ s^B \end{bmatrix} &= B^{-1}b - B^{-1}N \begin{bmatrix} x^N \\ s^N \end{bmatrix} \\ &= \begin{bmatrix} (B^t)^{-1}b^t \\ b^l - B^l(B^t)^{-1}b^t \end{bmatrix} - \begin{bmatrix} (B^t)^{-1}N^t & (B^t)^{-1} \\ N^l - B^l(B^t)^{-1}N^t & -B^l(B^t)^{-1} \end{bmatrix} \begin{bmatrix} x^N \\ s^N \end{bmatrix} \end{aligned} \tag{2}$$

and

$$\begin{aligned} z &= \begin{bmatrix} c^B & 0 \end{bmatrix} \begin{bmatrix} x^B & s^B \end{bmatrix}^\top + \begin{bmatrix} c^N & 0 \end{bmatrix} \begin{bmatrix} x^N & s^N \end{bmatrix}^\top \\ &= \begin{bmatrix} c^B & 0 \end{bmatrix} B^{-1}b + \left(\begin{bmatrix} c^N & 0 \end{bmatrix} - \begin{bmatrix} c^B & 0 \end{bmatrix} B^{-1}N\right) \begin{bmatrix} x^N & s^N \end{bmatrix}^\top \\ &= c^B(B^t)^{-1}b^t + \begin{bmatrix} c^N - c^B(B^t)^{-1}N^t & -c^B(B^t)^{-1} \end{bmatrix} \begin{bmatrix} x^N & s^N \end{bmatrix}^\top \end{aligned} \tag{3}$$

The quantities $\begin{bmatrix} x^B & s^B \end{bmatrix}^\top = \overline{b} = B^{-1}b$ are the values of the basic variables, the quantities $y = \begin{bmatrix} c^B & 0 \end{bmatrix} B^{-1}$ are the dual variables, and the quantities $\overline{c} = \left(\begin{bmatrix} c^N & 0 \end{bmatrix} - \begin{bmatrix} c^B & 0 \end{bmatrix} B^{-1}N\right)$ are the reduced costs. A row or column of $B^{-1}N$ (as appropriate to the context) will be denoted $\overline{a}_k$ (the single subscript distinguishes it from an individual element $\overline{a}_{ij}$). A row or column of B^{-1} (as appropriate to the context) will be denoted β_k. When discussing pivot selection calculations, Δ_j will be the change in nonbasic variable x_j or s_j.

The dual problem is formed by first converting (1) to max $-cx$, giving

$$\begin{aligned} \min\ & yb \\ & yA \geq -c \\ & y \geq 0 \end{aligned}$$

Add surplus variables σ and partition $\begin{bmatrix} A & -I \end{bmatrix}^\top$ into basic and nonbasic portions as

$$\begin{bmatrix} \mathcal{B} \\ \mathcal{N} \end{bmatrix} = \left[\begin{array}{cc} 0 & -I^{\mathcal{B}} \\ B^t & N^t \\ \hline -I^{\mathcal{N}} & 0 \\ B^l & N^l \end{array}\right]$$

with corresponding partitions $\begin{bmatrix}\sigma^{\mathcal{B}} & y^{\mathcal{B}} & \sigma^{\mathcal{N}} & y^{\mathcal{N}}\end{bmatrix}$ for y, σ, and $\begin{bmatrix}c^B & c^N\end{bmatrix}$ for c. The right-hand side b is augmented with 0's in the rows corresponding to the surplus variables and partitioned as $\begin{bmatrix}0 & b^t & 0 & b^l\end{bmatrix}^\top$. The basis inverse will be

$$\mathcal{B}^{-1} = \begin{bmatrix}(B^t)^{-1}N^t & (B^t)^{-1} \\ -I^{\mathcal{B}} & 0\end{bmatrix}.$$

We then have

$$\begin{aligned}\begin{bmatrix}\sigma^{\mathcal{B}} & y^{\mathcal{B}}\end{bmatrix} &= (-c)\mathcal{B}^{-1} - \begin{bmatrix}\sigma^{\mathcal{N}} & y^{\mathcal{N}}\end{bmatrix}\mathcal{N}\mathcal{B}^{-1} \\ &= \begin{bmatrix}c^N - c^B(B^t)^{-1}N^t & -c^B(B^t)^{-1}\end{bmatrix} - \begin{bmatrix}\sigma^{\mathcal{N}} & y^{\mathcal{N}}\end{bmatrix}\begin{bmatrix}-(B^t)^{-1}N^t & -(B^t)^{-1} \\ B^l(B^t)^{-1}N^t - N^l & B^l(B^t)^{-1}\end{bmatrix}\end{aligned} \tag{4}$$

and

$$\begin{aligned}z &= \begin{bmatrix}\sigma^{\mathcal{B}} & y^{\mathcal{B}}\end{bmatrix}\begin{bmatrix}0 & b^t\end{bmatrix}^\top + \begin{bmatrix}\sigma^{\mathcal{N}} & y^{\mathcal{N}}\end{bmatrix}\begin{bmatrix}0 & b^l\end{bmatrix}^\top \\ &= (-c)\mathcal{B}^{-1}b^{\mathcal{B}} + \begin{bmatrix}\sigma^{\mathcal{N}} & y^{\mathcal{N}}\end{bmatrix}(b^{\mathcal{N}} - \mathcal{N}\mathcal{B}^{-1}b^{\mathcal{B}}) \\ &= -c^B(B^t)^{-1}b^t + \begin{bmatrix}\sigma^{\mathcal{N}} & y^{\mathcal{N}}\end{bmatrix}\begin{bmatrix}(B^t)^{-1}b^t \\ b^l - B^l(B^t)^{-1}b^t\end{bmatrix}\end{aligned} \tag{5}$$

When discussing pivot selection calculations, δ_j will be the change in nonbasic dual variable y_j or σ_j.

Let $e_k \in R^d$ be a row or column vector of appropriate dimension (as determined by the context), with a 1 in position k and 0's in all other positions.

3 Updating Formulæ

For purposes of the updating formulæ, the distinction between original variables x and slack variables s is not important. For simplicity, x_k is used to represent both original variables and slack variables in this section. In the same vein, c^B and c^N will denote $\begin{bmatrix} c^B & 0 \end{bmatrix}$ and $\begin{bmatrix} c^N & 0 \end{bmatrix}$, respectively.

3.1 Basis Updates

While these formulæ are not applied directly to update the basis, they are useful in deriving update formulæ for other values.

Suppose that x_i will leave basis position k and be replaced by x_j. The new basis B' can be expressed as $B' = B - a_i e_k + a_j e_k$. Premultiplying by B^{-1} and postmultiplying by $(B')^{-1}$, we have

$$\begin{aligned} B^{-1}B'(B')^{-1} &= B^{-1}B(B')^{-1} - B^{-1}a_i e_k (B')^{-1} + B^{-1}a_j e_k (B')^{-1} \\ B^{-1} &= (B')^{-1} - \overline{a}_i \beta'_k + \overline{a}_j \beta'_k \\ (B')^{-1} &= B^{-1} + \overline{a}_i \beta'_k - \overline{a}_j \beta'_k \end{aligned} \tag{6}$$

Since x_i was basic, $\overline{a}_i = e_k$. This gives

$$(B')^{-1} = B^{-1} + e_k \beta'_k - \overline{a}_j \beta'_k.$$

Premultiplying by e_l to obtain an update formula for row l, we have

$$\begin{aligned} \beta'_l &= \beta_l - \frac{\overline{a}_{lj}}{\overline{a}_{kj}} \beta_k \qquad l \neq k \\ \beta'_k &= \frac{1}{\overline{a}_{kj}} \beta_k \end{aligned} \tag{7}$$

3.2 Primal Variable Updates

Updating the primal variables is straightforward and follows directly from (2).

Both primal and dual pivots calculate the change in the entering primal variable, Δ_j. The entering variable x_j is set to $u_j + \Delta_j$ or $l_j + \Delta_j$, for x_j entering from its upper or lower bound, respectively. The leaving variable x_i is set to u_i or l_i, for x_i leaving at its upper or lower bound, respectively. The remaining basic variables x_k, $k \neq i$, are updated according to the formula

$$x_k = \overline{b}_k - \overline{a}_{kj} \Delta_j.$$

3.3 Dual Variable Updates

Updating the dual variables is simple in the final implementation, but a little work is necessary to derive the updating formula. The difficulty lies in the fact that the dual variables of interest are $y = \begin{bmatrix} y^{\mathcal{B}} & y^{\mathcal{N}} \end{bmatrix}$, *i.e.*, a mixture of basic and nonbasic dual variables. Direct application of (4) is not possible.

Assume that the leaving variable x_i occupies row k in the basis B. The new vector of basic costs, $(c')^B$, can be expressed as $(c')^B = c^B - [0 \dots c_i \dots 0] + [0 \dots c_j \dots 0]$, where c_i and c_j occur in the k^{th} position. From (6), it is easy to show $B(B')^{-1} = I + a_i(\beta')_k - a_j(\beta')_k$.

We can proceed to derive the update formulæ for y as follows:

$$\begin{aligned} y' &= (c')^B(B')^{-1} \\ &= c^B B^{-1} B(B')^{-1} - c_i(\beta')_k + c_j(\beta')_k \\ &= y(I + a_i(\beta')_k - a_j(\beta')_k) - c_i(\beta')_k + c_j(\beta')_k \\ &= y + (c_j - ya_j)(\beta')_k - (c_i - ya_i)(\beta')_k. \end{aligned}$$

Recognising that $\overline{c}_j = c_j - ya_j$ is the reduced cost of x_j before the basis change, and noting that $\overline{c}_i = c_i - ya_i = 0$ since x_i was basic, we have

$$y' = y + \overline{c}_j(\beta')_k.$$

As a further observation, note that $(\beta')_k = \beta_k/\overline{a}_{kj}$, so we can update y using a row of B^{-1} as

$$y' = y + \overline{c}_j\beta_k/\overline{a}_{kj}.$$

4 Pricing Algorithms

4.1 Projected Steepest Edge Pricing

The primal simplex algorithm in DYLP uses projected steepest edge (PSE) pricing; the algorithm used is described as dynamic projected steepest edge ('dynamic') in Forrest and Goldfarb [3].

To understand the operation of projected steepest edge (PSE) pricing, it will be helpful to start with the definition of a direction of motion. The values of the basic and nonbasic variables can be expressed as

$$\begin{bmatrix} x^B \\ x^N \end{bmatrix} = \begin{bmatrix} b \\ l/u \end{bmatrix} - \begin{bmatrix} B^{-1}A^N \\ -I \end{bmatrix} \Delta$$

where l/u is intended to indicate use of the lower or upper bound as appropriate for the particular nonbasic variable. When a given nonbasic variable x_j is moved by an amount Δ_j, the values of x will change as

$$-\begin{bmatrix} B^{-1}a_j \\ -e_j \end{bmatrix} \Delta_j = -\begin{bmatrix} \overline{a}_j \\ -e_j \end{bmatrix} \Delta_j = \eta_j \Delta_j$$

The vector η_j is the direction of motion as x_j is changed; alternatively, it is the edge of the polyhedron which is traversed as x_j is changed. Let $\gamma_j = \|\eta_j\|$ be the norm of η_j.

For pricing, it can be immediately seen that $c\eta_j = c_j - c^B\overline{a}_j$ is the reduced cost $\overline{c}_j$. Dantzig pricing chooses an entering variable x_j such that $\overline{c}_j$ has appropriate sign and the largest magnitude over all reduced costs, but it can be misled by differences in scaling from one column to the next. Steepest edge (SE) pricing scales $\overline{c}_j$ by γ_j, choosing an entering variable x_j with $\overline{c}_j$ of appropriate sign and the largest $\left|\dfrac{c\eta_j}{\|\eta_j\|}\right|$, effectively calculating the change in objective value over a unit vector in the direction of motion. This gives a uniform pricing comparison, using the slope of the edge.

Projected steepest edge (PSE) pricing uses 'projected' column norms which are calculated using a vector $\tilde{\eta}_j$ which contains only the components of η_j included in a reference frame. Initially, this reference frame contains only the nonbasic variables, so that $\tilde{\gamma}_j = 1$ for all $x_j \in x^N$. In order to avoid calculating $\tilde{\gamma}_j$ from scratch each time a column must be priced, the norms are iteratively updated.

To derive the update formulæ for $\tilde{\gamma}_j$, it is useful to start with the update formulæ for the full vector η_j. As mentioned in §3.3, for x_i leaving basis position k and x_j entering, $B(B')^{-1} = I + a_i(\beta')_k - a_j(\beta')_k$. Taking this one step further, $(B')^{-1} = B^{-1} + \overline{a}_i(\beta')_k - \overline{a}_j(\beta')_k$. Then for an arbitrary column a_p,

$$\begin{aligned} (B')^{-1}a_p &= B^{-1}a_p + \overline{a}_i(\beta')_k a_p - \overline{a}_j(\beta')_k a_p \\ \overline{a}'_p &= \overline{a}_p + e_k(\frac{\overline{a}_{kp}}{\overline{a}_{kj}}) - \overline{a}_j(\frac{\overline{a}_{kp}}{\overline{a}_{kj}}) \end{aligned} \tag{8}$$

(recalling that $(\beta')_k = \beta_k/\overline{a}_{kj}$).

To see that (8) amounts to $\eta'_p = \eta_p - \eta_j(\dfrac{\overline{a}_{kp}}{\overline{a}_{kj}})$, it's helpful to expand the vectors:

$$\overline{a}'_p = \begin{bmatrix} \overline{a}_{1p} \\ \vdots \\ \overline{a}_{kp} \\ \vdots \\ \overline{a}_{mp} \end{bmatrix} + \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \frac{\overline{a}_{kp}}{\overline{a}_{kj}} - \begin{bmatrix} \overline{a}_{1j} \\ \vdots \\ \overline{a}_{kj} \\ \vdots \\ \overline{a}_{mj} \end{bmatrix} \frac{\overline{a}_{kp}}{\overline{a}_{kj}}.$$

With a little thought, it can be seen that the middle term represents one half of the permutation which moves x_j into the basic partition of η'_j. (The other half moves x_i into the nonbasic partition). When updating η_i, the update formula can be collapsed to $\eta'_i = -\eta_j/\overline{a}_{kj}$, since $\overline{a}_{ki} = 1$. Summarising, the update formulæ for the edge directions η_j are

$$
\begin{aligned}
\eta'_p &= \eta_p - \eta_j(\frac{\overline{a}_{kp}}{\overline{a}_{kj}}), \qquad p \neq i \\
\eta'_i &= -\eta_j/\overline{a}_{kj}.
\end{aligned}
\tag{9}
$$

In fact, the code actually stores and updates γ_j^2. With (9) in hand, derivation of the update formulæ are straightforward:

$$
\begin{aligned}
(\gamma'_p)^2 &= \eta'_p \cdot \eta'_p \\
&= (\eta_p - \eta_j(\frac{\overline{a}_{kp}}{\overline{a}_{kj}})) \cdot (\eta_p - \eta_j(\frac{\overline{a}_{kp}}{\overline{a}_{kj}})) \\
&= \eta_p \cdot \eta_p - 2(\frac{\overline{a}_{kp}}{\overline{a}_{kj}})\eta_j \cdot \eta_p + (\frac{\overline{a}_{kp}}{\overline{a}_{kj}})^2 \eta_j \cdot \eta_j \\
&= \gamma_p^2 - 2(\frac{\overline{a}_{kp}}{\overline{a}_{kj}}) \begin{bmatrix} \overline{a}_j^T & e_j^T \end{bmatrix} \begin{bmatrix} \overline{a}_p \\ e_p \end{bmatrix} + (\frac{\overline{a}_{kp}}{\overline{a}_{kj}})^2 \gamma_j^2 \\
&= \gamma_p^2 - 2(\frac{\overline{a}_{kp}}{\overline{a}_{kj}})(\overline{a}_j^T B^{-1}) a_p + (\frac{\overline{a}_{kp}}{\overline{a}_{kj}})^2 \gamma_j^2
\end{aligned}
\tag{10}
$$

$$
\begin{aligned}
(\gamma'_i)^2 &= \eta'_i \cdot \eta'_i \\
&= \eta_j/\overline{a}_{kj} \cdot \eta_j/\overline{a}_{kj} \\
&= \gamma_j^2/\overline{a}_{kj}^2
\end{aligned}
\tag{11}
$$

Equations (9) can be used directly to update the $\tilde{\eta}_j$. To adapt (10) and (11) for the $\tilde{\gamma}_j$, a little algebra should serve to see that it's sufficient to substitute $\tilde{a}_j$ in (10), as well as using $\tilde{\gamma}_p$ and $\tilde{\gamma}_j$.

It is straightforward to observe that when equations (9) are premultiplied by c, they can be used to update the reduced costs as

$$
\begin{aligned}
\overline{c}'_p &= \overline{c}_p - \overline{c}_j(\frac{\overline{a}_{kp}}{\overline{a}_{kj}}) \qquad p \neq i \\
\overline{c}'_i &= -\overline{c}_j/\overline{a}_{kj}.
\end{aligned}
$$

4.2 Dual Steepest Edge Pricing

The dual simplex in DYLP uses dual steepest edge (DSE) pricing; the algorithm used is described as dual algorithm 1 ('steepest 1') in Forrest and Goldfarb [3].

The values $\overline{b} = B^{-1}b$ are the reduced costs of the nonbasic dual variables. Analogous to Dantzig pricing in the primal case, one can choose a entering dual variable y_i such that $\overline{b}_i$ has appropriate sign and the largest magnitude over all reduced costs, but there is the same problem with scaling. The version of dual steepest edge (DSE) pricing implemented in DYLP scales $\overline{b}_i$ by $\rho_i = \|\beta_i\|$, choosing a leaving variable x_i with $\overline{b}_i$ of appropriate sign and the largest $\left|\frac{\beta_i b}{\|\beta_i\|}\right|$, effectively calculating the change in the dual objective value over a unit vector in the dual direction of motion in the space of the dual variables. This gives a uniform pricing comparison, using the slope of the dual edge.

In the next few paragraphs, an alternative motivation of the algorithm is presented which (perhaps) clarifies the relationship between dual algorithm 1 and dual algorithm 2 in that paper[1].

To see how DSE operates within the context of the revised primal simplex tableau, we can refer back to equations (4) and (5) from §2, repeated here:

$$\begin{aligned}\begin{bmatrix}\sigma^{\mathcal{B}} & y^{\mathcal{B}}\end{bmatrix} &= (-c)\mathcal{B}^{-1} - \begin{bmatrix}\sigma^{\mathcal{N}} & y^{\mathcal{N}}\end{bmatrix}\mathcal{N}\mathcal{B}^{-1} \\ &= \begin{bmatrix}c^N - c^B(B^t)^{-1}N^t & -c^B(B^t)^{-1}\end{bmatrix} - \begin{bmatrix}\sigma^{\mathcal{N}} & y^{\mathcal{N}}\end{bmatrix}\begin{bmatrix}-(B^t)^{-1}N^t & -(B^t)^{-1} \\ B^l(B^t)^{-1}N^t - N^l & B^l(B^t)^{-1}\end{bmatrix}\end{aligned} \tag{4}$$

and

$$\begin{aligned} z &= \begin{bmatrix}\sigma^{\mathcal{B}} & y^{\mathcal{B}}\end{bmatrix}\begin{bmatrix}0 & b^t\end{bmatrix}^T + \begin{bmatrix}\sigma^{\mathcal{N}} & y^{\mathcal{N}}\end{bmatrix}\begin{bmatrix}0 & b^l\end{bmatrix}^T \\ &= (-c)\mathcal{B}^{-1}b^{\mathcal{B}} + \begin{bmatrix}\sigma^{\mathcal{N}} & y^{\mathcal{N}}\end{bmatrix}(b^{\mathcal{N}} - \mathcal{N}\mathcal{B}^{-1}b^{\mathcal{B}}) \\ &= -c^B(B^t)^{-1}b^t + \begin{bmatrix}\sigma^{\mathcal{N}} & y^{\mathcal{N}}\end{bmatrix}\begin{bmatrix}(B^t)^{-1}b^t \\ b^l - B^l(B^t)^{-1}b^t\end{bmatrix}\end{aligned} \tag{5}$$

Recall that the values of the dual basic variables are the reduced costs of the primal problem, and the reduced costs of the dual variables are the values of the primal basic variables (*cf.* equations (2) and (3)).

By analogy to the primal pivoting rules, for dual simplex we want to choose a nonbasic dual variable which will move us in a direction of steepest descent. If the nonbasic dual is to increase, its reduced cost must be less than 0 in order to see a reduction in the dual objective. This corresponds to the case of a primal variable which will be increased and driven out of the basis at its lower bound with a positive primal reduced cost. If the nonbasic dual is to decrease, its reduced cost must be greater than 0 in order to see a reduction in the dual objective. This corresponds to the case of a primal variable which will be decreased and driven out of the basis at its upper bound with a negative primal reduced cost.

The actual direction of motion in the full dual space (y and σ) would be specified by a row of

$$\mathcal{N}\mathcal{B}^{-1} = \begin{bmatrix}-(B^t)^{-1}N^t & -(B^t)^{-1} \\ B^l(B^t)^{-1}N^t - N^l & B^l(B^t)^{-1}\end{bmatrix},$$

a vector which is not readily available in the revised primal simplex. (Moreover, for the typical problem in which the number of variables greatly exceeds the number of constraints, the norm of this vector is expensive to calculate when initialising the pricing algorithm, and the updates are expensive. This is the algorithm which Forrest and Goldfarb describe as dual algorithm 2.)

However, one can make an argument that there's no need to consider the component of the direction of motion in the subspace of the dual surplus variables. (More positively, we can take the view that we're only interested in motion in the polyhedron $\{y \in R^m \mid yA \geq -c, y \geq 0\}$ defined by the dual variables.) Changes in the surplus variables cannot affect the objective directly, as they account for the 0's in the augmented and partitioned b vector. Algebraically, we can see that the dual basic portion of b, $\begin{bmatrix}0 & b^t\end{bmatrix}^T$, guarantees that there will never be any contribution from the columns of $\mathcal{N}\mathcal{B}^{-1}$ involving N. The component of motion in the space of the dual variables y is

[1]Those who have read [3] are warned that the author's notation is in no way compatible with that of Forrest and Goldfarb.

then simply the rows β_l of B^{-1}, which are easily available from the primal tableau. (The analogous action in the primal problem — ignore the component of η_j in the subspace of the primal slack variables — offers no computational advantage.)

Given a rationale for taking the rows β_l of B^{-1} as the edges of interest, what remains is to work out the details. Since we're aiming for a steepest edge algorithm, we'll be interested in iteratively updating $\|\beta_l\|^2 = \beta_l \cdot \beta_l$, the square of the norm of a row β_l. Given the update formulæ for β_l derived in §3.1, the development of the update formulæ for $\rho_l = \|\beta_l\|^2$ is straightforward algebra. Let x_i be the leaving variable and x_j be the entering variable, and assume x_i occupies row k of the basis B before the update. We have

$$\begin{aligned} \rho'_l &= \rho_l - 2\frac{\overline{a}_{lj}}{\overline{a}_{kj}}\beta_l \cdot \beta_k + (\frac{\overline{a}_{lj}}{\overline{a}_{kj}})^2\rho_k \qquad\qquad l \neq k \\ \rho'_k &= (\frac{1}{\overline{a}_{kj}})^2\rho_k \end{aligned} \tag{12}$$

Since the update will be performed for all rows in the basis, it's worth calculating the vector $\tau = B^{-1}\beta_k^T$ to obtain all the inner products $\beta_l \cdot \beta_k$ in one calculation.

5 Anti-Degeneracy Using a Perturbed Subproblem

In both primal and dual simplex, DYLP implements an anti-degeneracy algorithm using a perturbed subproblem. It builds on a method described by Ryan & Osborne [10] in which all variables are assumed to have lower bounds of zero and upper bounds of infinity.

The original algorithm is easily described in terms of the primal problem. When degeneracy is detected, a restricted subproblem is formed consisting only of the constraints involved in the degeneracy (*i.e.*, constraints i such that $\overline{b}_i = 0$). The values $\overline{b}_i$ are given (relatively) large perturbations and pivots are performed within the context of the restricted subproblem until a direction of recession from the degenerate vertex is found (indicated by apparent unboundedness). The original unperturbed values of $\overline{b}_i$ are then restored (since all pivots were, in actuality, simply changes of basis while remaining at the degenerate vertex) and the full problem is resumed.

An alternative view goes directly back to the constraints involved in the degeneracy. By perturbing their right-hand-side values b_i, the single vertex formed by the constraints is fractured into many vertices. For the simple case of $0 \leq x \leq \infty$, we have $\overline{b} = B^{-1}b$, so perturbing $\overline{b}$ by the vector ξ is equivalent to perturbing b by the vector $-B\xi$.

In dual simplex, this algorithm can be implemented directly. The restricted subproblem is formed from the dual constraints (primal columns) corresponding to basic dual variables (primal reduced costs) whose value is zero. The perturbation is introduced directly to the values $\overline{c}_j$, taking care to maintain dual feasibility. The perturbation is maintained by the incremental update of the dual variables and reduced costs after each pivot. When accuracy checks are performed, the correct value of zero can be substituted on the fly for the perturbed values.

The trick to implementing this algorithm in the context of variables with arbitrary upper and lower bounds is to distinguish between apparent motion due to the introduced perturbations and real motion (along a direction of recession) which is nonetheless limited by a bound on a variable. DYLP uses an array, dy_brkout, to record the direction of change (away from the current bound) required for nondegenerate but bounded motion.

A second, more subtle problem, is that the perturbation for a given variable must be sufficiently small to avoid a false indication of a nondegenerate pivot. DYLP scales the perturbation to be at most $.001(u_i - l_i)$, but there is no easy way to guarantee that this is sufficiently small. Consider two variables x_i and x_k, and assume that they occupy rows i and k in the basis, with perturbed values $\tilde{b}_i$ and $\tilde{b}_k$, respectively. For concreteness, assume that each was originally degenerate at its lower bound, so that a pivot which resulted in one variable leaving at its upper bound would be nondegenerate. For $\overline{a}_{ij}$ and $\overline{a}_{kj}$ of appropriate sign to move x_i toward l_i and x_k toward u_k, given a situation where $|\overline{a}_{ij}| \ll |\overline{a}_{kj}|$, it is not possible to assure that

$$\frac{\tilde{b}_i - l_i}{\overline{a}_{ij}} < \frac{u_k - \tilde{b}_k}{\overline{a}_{kj}}$$

without actually testing each pair. In this case, the perturbation introduced for x_i is too large, and the resulting Δ_{ij} *appears* to allow x_k to become the limiting variable, leaving the basis with a bounded but nondegenerate change. When DYLP detects this problem, it will reduce the perturbation by a factor of 10 and form the restricted subproblem again. If a (small) limit on the number of attempts is exceeded, DYLP simply gives up and takes a degenerate pivot.

A second problem occurs when a perturbation is so small as to be indistinguishable next to the bound. Specifically, the test to determine if a variable x_i is at bound is dy_tols.zero$(1 + |bnd_i|) < |x_i - bnd_i|$. If bnd_i is large, the perturbation can be swamped. This situation can arise if u_i and

l_i as given to DYLP are nearly equal, or due to reduction of the perturbation as described in the previous paragraph.

6 Lightweight Anti-Degeneracy Measures Based on Hyperplane Alignment

In addition to the perturbed subproblem anti-degeneracy algorithm described in §5, DYLP provides a light-weight anti-degeneracy mechanism based on hyperplane alignment. In the code and documentation, this is referred to as 'anti-degen lite'.

Each constraint $a_k x \leq b_k$ defines an associated hyperplane at equality. In the absence of degeneracy, a simplex pivot consists of moving away from one hyperplane along an edge until another hyperplane blocks further progress. The hyperplane being left becomes loose, and the blocking hyperplane becomes tight. The choice of entering variable x_j determines the constraint that will become loose, and the choice of leaving variable x_i determines the constraint that will become tight.

Ideally, the choice of constraints is unique, but life is seldom ideal. Most often the lack of uniqueness is due to degeneracy, in which one or more basic variables are at their upper or lower bounds. Geometrically, there are more tight constraints than required to define the current extreme point. In this case the change of basis that occurs with the pivot will not result in a move to a new extreme point.

This section describes a suite of measures based on hyperplane alignment which try to better the odds of selecting hyperplanes which will form an edge that escapes from the degenerate extreme point.

Because all constraints at a degenerate vertex are tight, some terminology will be useful to describe the changes associated with a pivot. For this section only, the terms activate and deactivate will be used as follows:

- When the slack variable for a constraint moves to the basic partition, the constraint is deactivated. When the slack variable moves to the nonbasic partition, the constraint is activated.
- When an architectural variable moves to the basic partition, the relevant bound constraint is deactivated. When an architectural variable moves to the nonbasic partition, the relevant bound constraint is activated.

6.1 Activation of Constraints

In both the primal and dual simplex algorithms, the constraint which is activated by a pivot depends on the leaving variable and its direction of motion. Before discussing the types of alignment calculations, it will be useful to discuss the activation of constraints. Knowing the type of constraint ('≤' or '≥') is necessary because it determines the direction of the normal with respect to the feasible region.

DYLP assumes that the majority of explicit constraints of the primal problem are of the form $a_k x \leq b_k$. It also understands range constraints of the form $\check{b}_k \leq a_k x \leq b_k$. These are implemented by placing an upper bound on the associated slack variable s_k, but for purposes of determining the constraint to be activated we need to recognise that there are really two constraints, $a_k x \geq \check{b}_k$ and $a_k x \leq b_k$.

Bounded variables are handled implicitly by the primal simplex algorithm. When a bounded variable becomes nonbasic at its lower bound, the constraint $x_k \geq l_k$ is activated; when it becomes nonbasic at its upper bound, the constraint $x_k \leq u_k$ is activated.

A final complication is introduced in phase I of the primal simplex, where it's possible to approach a constraint from the 'wrong' side in the process of finding a primal feasible basic solution. For example, if a slack variable $s_k < 0$ will increase and leave the basis at 0, the constraint which is becoming tight is actually $a_k x \geq b_k$. « *Is this really a valid insight? In terms of blocking motion, it's true. In terms of alignment with the objective, for example, I have doubts.* »

Turning to the dual problem, the question of what constraint is being activated is substantially obscured by the mechanics of running the dual simplex algorithm from the primal data structures. A much clearer picture can be obtained by expanding the primal system to include explicit upper and lower bound constraints and examining the resulting dual constraints ([3, §3.4], or see [4] for an extended development). Briefly, let y be the dual variables associated with the original explicit constraints $a_k x \leq b_k$ (the architectural constraints), $\check{y}$ be the dual variables associated with the lower bound constraints, and $\hat{y}$ be the dual variables associated with the upper bound constraints. A superscript $\underline{N}$ will represent the set of primal variables at their lower bound, $\overline{N}$ the set of primal variables at their upper bound, and B the set of basic primal variables. The set of dual constraints can then be written as

$$\begin{aligned} yB - \check{y}^B I + \hat{y}^B I &= c^B \\ y\underline{N} - \check{y}^{\underline{N}} I + \hat{y}^{\underline{N}} I &= c^{\underline{N}} \\ y\overline{N} - \check{y}^{\overline{N}} I + \hat{y}^{\overline{N}} I &= c^{\overline{N}} \end{aligned}$$

where the first term in each dual constraint comes from the primal architectural constraints, the second term from the lower bound constraints, and the third term from the upper bound constraints. The variables $\check{y}^B$, $\hat{y}^B$, $\hat{y}^{\underline{N}}$, and $\check{y}^{\overline{N}}$ are dual nonbasic and therefore have the value zero. (They are associated with primal bound constraints which are not tight.) We can rewrite the dual constraints as

$$\begin{aligned} yB &= c^B \\ y\underline{N} - \check{y}^{\underline{N}} I &= c^{\underline{N}} \\ y\overline{N} + \hat{y}^{\overline{N}} I &= c^{\overline{N}} \end{aligned}$$

We can then interpret the constraints $y\underline{N} - \check{y}^{\underline{N}} I = c^{\underline{N}}$ as $y\underline{N} \geq c^{\underline{N}}$, with $\check{y}^{\underline{N}}$ acting as the surplus variables. Similarly, the constraints $y\overline{N} + \hat{y}^{\overline{N}} I = c^{\overline{N}}$ can be interpreted as $y\overline{N} \leq c^{\overline{N}}$, with $\hat{y}^{\overline{N}}$ acting as the slack variables.

With this interpretation in hand, it's easy to determine the hyperplane that's activated by a pivot. When a dual variable $\check{y}_k^{\underline{N}}$ is driven out of the basis at 0 (x_k enters rising from its lower bound), the constraint $ya_k \geq c_k$ becomes tight. When a dual variable $\hat{y}_k^{\overline{N}}$ is driven out of the basis at 0 (x_k enters decreasing from its upper bound), the constraint $ya_k \leq c_k$ becomes tight. This interpretation is uniform for the original primal variables as well as the primal slack variables.

For the most common case of a primal constraint $a_i x \leq b_i$, with associated slack s_i, $0 \leq s_i \leq \infty$, the dual constraint reduces to $y_i \geq 0$, and this is handled as an implicit bound by the dual simplex algorithm implemented in DYLP. (Range constraints complicate the interpretation, but not the mechanics, of the implementation. Again, see [4] for a detailed explanation.)

In the sections which follow, the alignment calculations are developed in terms of the most common constraint form ($a_k x \leq b_k$ in the case of the primal simplex, and $ya_k \geq c_k$ in the case of the dual simplex). Accommodating the different constraint types described in this section is simply a matter of correcting the sign of the calculation as needed to account for the direction of the constraint normal.

6.2 Alignment With Respect to the Objective Function

The primal objective used in DYLP is $\min cx$. We need to move in the direction $-c$ until we reach an extreme point of the polytope where the cone formed by the normals of the active constraints includes $-c$.

If the goal is to travel in the direction $-c$, one approach would be to leave each vertex by moving along the edge which most nearly points in the direction $-c$. The edges traversed by the simplex algorithm are simply the intersections of active hyperplanes. If we're trying to construct an edge with which we can leave a degenerate vertex, we could choose to activate a hyperplane $a_k x = b_k$ such that $-c$ most nearly lies in the hyperplane, on the theory that its intersection with other active hyperplanes at the vertex is more likely to produce an edge with the desired orientation. This is the 'Aligned' strategy, because we want the hyperplanes most closely aligned with the normal of the objective.

Going to the other extreme, at the optimal vertex it must be true that the active hyperplanes block further motion in the direction $-c$, and $-c$ must lie within the cone of normals of the active hyperplanes. One can make the argument that a good choice of hyperplane would the one that most nearly blocks motion in the direction $-c$, as it's likely to be active at the optimal vertex. This is called the 'Perpendicular' strategy, because we want the hyperplanes which are most nearly perpendicular to the normal of the objective.

For constraints $a_k x \leq b_k$ the normal points out of the feasible region. Let the alignment of the normal a_k with $-c$ be calculated as $\dfrac{a_i \cdot c}{\|a_i\|\|c\|}$. Then for the Perpendicular strategy, we want to select the hyperplane $a_i x = b_i$ such that $i = \arg\max_k \dfrac{a_k \cdot c}{\|a_k\|}$ over all constraints $a_k x \leq b_k$ in the degenerate set.

For the Aligned strategy, the criteria is a bit more subtle. If $a_k \cdot -c = 0$, $-c$ lies in the hyperplane $a_k x = b_k$. Selecting the hyperplane i such that $i = \arg\min_k \left|\dfrac{a_k \cdot c}{\|a_k\|}\right|$ is not quite sufficient. Where possible, DYLP attempts to choose hyperplanes which are tilted in the direction of the objective, so as to bound the problem. The preferred hyperplane is $a_i x = b_i$ such that $i = \arg \min_{\{k | a_k \cdot c \geq 0\}} \dfrac{a_k \cdot c}{\|a_k\|}$ over the constraints in the degenerate set. If $a_k \cdot c < 0$ for all k, the preferred hyperplane is chosen as $i = \arg\max_k \dfrac{a_k \cdot c}{\|a_k\|}$.

The dual objective used in DYLP is $\min yb$, but we must be careful here to to include the effect of the bounds on the primal variables. The objective is properly stated as $\min \begin{bmatrix} y & \check{y} & \hat{y} \end{bmatrix} \begin{bmatrix} b & -l & u \end{bmatrix}^\top$, and we will need to include the coefficients of $\check{y}$ and $\hat{y}$ in the constraint normals. (In the primal we could ignore this effect, because the objective coefficients associated with the slack variables are uniformly zero.)

For dual constraints $ya_k \geq c_k$, the normal $\begin{bmatrix} a_k & -e_k & 0 \end{bmatrix}$ will point into the feasible region and DYLP calculates the alignment of $\begin{bmatrix} -b & l & -u \end{bmatrix}$ with the hyperplane as $\dfrac{b \cdot a_k + l_k}{(\|a_k\| + 1)\|\begin{bmatrix} b & -l & u \end{bmatrix}\|}$, so that a positive result identifies a constraint which blocks motion in the direction of the objective. For a constraint $ya_k \leq c_k$, the calculation is $\dfrac{(-b) \cdot a_k - u_k}{(\|a_k\| + 1)\|\begin{bmatrix} b & -l & u \end{bmatrix}\|}$. Selection of a specific leaving variable $\check{y}_k^N$ or $\hat{y}_k^{\bar{N}}$ is done using the same criteria outlined for the Perpendicular and Aligned cases in the primal problem.

6.3 Alignment With Respect to the Direction of Motion

The selection of an entering variable specifies the desired direction of motion for the pivot. At a degenerate vertex, we cannot move in the desired direction because the set of active hyperplanes does not contain this edge. Intuitively, activating a hyperplane which is closely aligned with the desired direction of motion might increase the chance of being able to move in that direction.

For the primal simplex, the direction of motion derived in §4.1 is $\eta_j = \begin{bmatrix} -B^{-1}a_j & -e_j \end{bmatrix}^\top$. The normal of a constraint $a_k x \leq b_k$ points out of the feasible region. The alignment of η_j and the normal a_k is calculated as $\dfrac{a_k \cdot \eta_j}{\|a_k\|\|\eta_j\|}$, so that a positive value identifies a hyperplane which blocks motion in the direction η_j.

It's important to note that normal a_k in this calculation is that of the inequality — the coefficient associated with the slack s_k is *not* included. This means that $a_k \cdot \eta_j \equiv -\overline{a}_{kj}$. For a bound constraint, the relation is obvious by inspection. If, for example, the constraint is $x_k \leq u_k$, the normal is e_k, and $e_k \cdot -\overline{a}_j = -\overline{a}_{kj}$. For an architectural constraint, it's necessary to look at the calculation in a way that separates the contributions of the architectural and slack variables, and basic and nonbasic variables. We are interested in the structure of the product $\begin{bmatrix} B & N \end{bmatrix} \begin{bmatrix} -B^{-1}N \\ I \end{bmatrix}$ for loose constraints which will be activated by pivoting the associated slack variable out of the basis. Breaking up the matrices as detailed in §2, we have

$$\begin{aligned}
\begin{bmatrix} B^l & I & N^l & 0 \end{bmatrix} \begin{bmatrix} -B^{-1}N \\ I \end{bmatrix} &= \begin{bmatrix} B^l & I & N^l & 0 \end{bmatrix} \begin{bmatrix} -\begin{bmatrix} (B^t)^{-1} & 0 \\ -B^l(B^t)^{-1} & I \end{bmatrix} \begin{bmatrix} N^t \\ N^l \end{bmatrix} \\ \begin{bmatrix} I & 0 \\ 0 & I \end{bmatrix} \end{bmatrix} \\
&= \begin{bmatrix} B^l & I & N^l & 0 \end{bmatrix} \begin{bmatrix} -(B^t)^{-1}N^t & -(B^t)^{-1} \\ B^l(B^t)^{-1}N^t - N^l & B^l(B^t)^{-1} \\ I & 0 \\ 0 & I \end{bmatrix} \\
&= \begin{bmatrix} -B^l(B^t)^{-1}N^t + B^l(B^t)^{-1}N^t - N^l + N^l & -B^l(B^t)^{-1} + B^l(B^t)^{-1} \end{bmatrix}
\end{aligned}$$

Removing the contribution due to the basic slack variables, we have $\begin{bmatrix} -B^l(B^t)^{-1}N^t + N^l & -B^l(B^t)^{-1} \end{bmatrix}$. Because the leaving variable for the pivot is a slack, the pivot element $\overline{a}_{kj}$ will be drawn from the component $\begin{bmatrix} B^l(B^t)^{-1}N^t - N^l & B^l(B^t)^{-1} \end{bmatrix}$ in $-B^{-1}N$, and the equivalence is verified.

To finish the alignment calculation for the purposes of selecting a leaving variable, all that is needed is to perform the normalisation by $\|a_k\|\|\eta_j\|$, and since $\|\eta_j\|$ is constant during the selection of the leaving variable, we need only divide by $\|a_k\|$ for comparison purposes. The selection of a leaving variable using the Aligned strategy is as outlined in the previous section.

Given that $a_k \cdot \eta_j \equiv -\overline{a}_{kj}$, it's worth taking a moment to consider a common tie-breaking rule for selecting the leaving variable — pick the variable with the largest $|\overline{a}_{kj}|$, to maintain numerical stability. In fact, this amounts to selecting a hyperplane to activate using an unnormalised variation of the Perpendicular strategy. The obvious corollary is that using the Aligned strategy presents a potential danger to numerical stability by deliberately choosing small pivots.

For the dual simplex, the direction of motion ζ_i is more complicated. Fortunately, we need only consider the portion of ζ_i in the space of the dual variables y. As derived in §4.2, this is simply row β_i of B^{-1}. For the dual constraints $ya_k \geq c_k$, the normal points into the feasible region. To maintain the convention that the alignment calculation should produce a positive result if the constraint blocks motion, the alignment calculation used by DYLP is $-\dfrac{\zeta_i \cdot a_k}{\|\zeta_i\|\|a_k\|}$. Given that we're only interested in the portion of $\zeta_i \cdot a_k$ contributed by the dual variables y, it's immediately apparent that the alignment calculation can be reduced to $-\frac{\overline{a}_{ik}}{\|a_k\|}$ for purposes of selecting the leaving dual variable. The final selection of a leaving dual variable using the Aligned or Perpendicular strategy proceeds as outlined in the previous section.

7 The LP Basis

DYLP requires three capabilities from a basis maintenance module:

- Factoring of the basis to create the basis inverse.
- Update of the basis inverse for a pivot.
- Premultiplication ('ftran') of a column vector by the basis inverse, and postmultiplication ('btran') of a row vector by the basis inverse.

DYLP uses the basis maintenance module from GLPK to provide these services. Knowledge of the structure and operation of the GLPK subroutines is confined to a set of interface subroutines in the file dy_basis.c. The majority of these are straightforward interface functions whose sole purpose is to hide the GLPK structures and to mediate between GLPK and the remainder of the code.

7.1 The GLPK Basis Module Interface

Very roughly, the GLPK basis maintenance module has a two-layer structure. The top layer (glpinv.c) provides the basic services for a generic basis inverse. In turn, the top layer calls on a second layer (glpluf.c) to provide a specific implementation of the basis inverse data structures and algorithms. Dynamic Markowitz pivoting with partial threshold pivot selection is used to factor a basis.

The routine dy_initbasis is used to initialise the basis module. The capacity of the basis, algorithm options, and numeric tolerances are set at initialisation (*vid.* §16.3). The basis is deleted by the routine dy_freebasis. Changing the basis capacity is implemented in DYLP by saving options and tolerances for the existing basis, deleting the existing basis, and creating a new basis of the appropriate size. The capacity is checked each time the basis is factored; changes are invisible to clients. The GLPK basis module will resize its own internal data structures whenever it determines that this is required.

In the main, DYLP uses the basis module in a standard way for factoring and pivoting. There are some departures from GLPK defaults:

- The initial size of the sparse vector working area is tripled.
- The limit on element growth (luf.max_gro) is reduced from 10^{12} to 10^{6}.
- The minimum value for elements on the diagonal of the factorisation (luf_basis.upd_tol is reduced from 10^{-6} to 10^{-10}.
- Instead of a fixed default of .1, the pivot stability tolerance is dynamically adjusted in a range between .01 and .95 based on DYLP's assessment of the numerical stability of the current basis. The number of pivot candidates examined when factoring the basis is also adjusted in the range 4 to 10. More candidates are considered as the stability requirement is raised in the hope of finding a numerically stable candidate without compromising sparsity.

The routine dy_setpivparms is provided to adjust the pivot stability tolerance and pivot candidate limit. Adjustment of the pivot selection parameters is done according to a fixed schedule of tolerance and limit values kept in the static data structure dy_basis.c:pivtols. The client specifies an integer delta which is used to select a pair of values from the schedule.

Pre- and post-multiplication of vectors by the basis inverse are provided by the routines dy_ftran and dy_btran, respectively.

7.2 Factoring

For factoring the basis, the routine dy_factor provides significant error recovery functions on top of the basic abilities of GLPK. The call structure is shown in Figure 1.

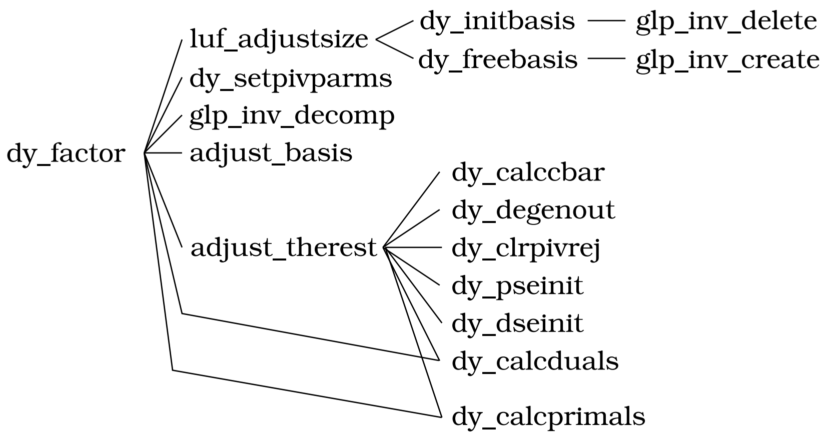

Figure 1: Call Graph for dy_factor

A singular basis can occur because of a simplex pivot attempt or as the result of a change in the coefficients of the basis because the client has fixed variables and then requested a warm or hot start. The factoring routine glp_inv_decomp detects a singular basis and reports the unpivoted rows and columns, but does not attempt to fix the basis. adjust_basis uses the information reported by glp_inv_decomp to attempt to patch the basis, substituting columns associated with slack variables for the set of columns identified as singular. This sequence is repeated until the basis is successfully factored.

In the larger context of DYLP, patching the basis is the least of the work. dy_factor will call adjust_therest to adjust the DYLP data structures as necessary to reflect the exchange of variables between the basic and nonbasic partitions. Depending on the phase, this can include updating the structures which maintain the basis, recalculating the primal (dy_calcprimals) and dual (dy_calcduals) variables, recalculating the reduced costs (dy_calccbar), resetting the DSE or PSE norms (dy_dseinit and dy_pseinit, respectively), clearing the list of variables marked ineligible for pivoting (dy_clrpivrej), and backing out a perturbed subproblem (dy_degenout).

glp_inv_decomp will abort an attempt to factor the basis if the current pivot selection parameters give rise to numerical instability (detected as excessive growth in the magnitude of the coefficients of the factored basis). dy_factor will make repeated tries to factor the basis, tightening the pivot selection parameters before each attempt. It will admit failure only if the numerical instability remains after the pivot selection tolerances have been tightened as much as possible, so that each pivot chosen is the maximum coefficient remaining in the unpivoted portion of the basis.

7.3 Pivoting

Pivoting is performed by dy_pivot, which confirms the numerical stability of the pivot element and calls glp_inv_update to pivot the basis.

To be judged numerically stable, a prospective pivot coefficient $\overline{a}_{ij}$ must exceed the product of the GLPK stability multiplier (luf.piv_tol), the DYLP pivot selection multiplier (dy_tols.pivot), and the maximum element in the transformed column $\overline{a}_j = B^{-1}a_j$ (primal simplex) or row $\overline{a}_i = \beta_i N$ (dual simplex). Standard defaults in DYLP are 5×10^{-2} for the GLPK stability multiplier and 1×10^{-5} for the DYLP pivot selection multiplier, so that the pivot coefficient is required to satisfy $|\overline{a}_{ij}| > (5 \times 10^{-7})(\max_k |\overline{a}_{kj}|)$ (primal simplex) or $|\overline{a}_{ij}| > (5 \times 10^{-7})(\max_k |\overline{a}_{ik}|)$ (dual simplex). The routine dy_chkpiv is supplied to perform this test, and is used as a qualification test by the routines which select the leaving primal variable in primal simplex and the entering primal variable in dual simplex. The check performed in dy_pivot should not fail, but is retained as a precaution.

If $\overline{a}_{ij}$ is rejected as numerically unstable, the pivot attempt is aborted. In primal simplex, the entering variable x_j will be placed on the rejected pivot list. For dual simplex, the leaving variable x_i is placed on the rejected pivot list. Recovery from pivoting problems and the handling of the rejected pivot list are discussed in §11.2.

A pivot can also fail if it results in a singular basis or if the basis representation runs out of space. The implementation of GLPK requires that the basis be reloaded and factored to recover from these errors; this is orchestrated by dy_duenna and discussed in §11.2.

Note that glp_inv_update expects to be supplied with $L^{-1}a_j$ as a hidden parameter. GLPK provides the capability to control whether a call to glp_inv_ftran sets this hidden parameter. This capability is exposed to clients as the second parameter to dy_ftran.

8 Accuracy Checks and Maintenance

Primal and dual accuracy checks, primal and dual feasibility checks, and factoring of the basis can be requested through the routine dy_accchk; each action can be requested separately.

DYLP refactors the basis and performs accuracy checks at regular intervals, based on a count of pivots which actually change the basis. By default, primal and dual accuracy checks are performed at twice this frequency. During phase II of the primal and dual simplex algorithms, the appropriate feasibility check is performed following each accuracy check. dy_duenna tracks the pivot count and requests checks and factoring at the scheduled intervals.

dy_accchk uses dy_factor to factor the basis and recalculate the primal and dual variables. When the basis has been factored and has passed the accuracy checks, the routine groombasis checks that the status of the basic variables matches their values and makes any necessary adjustments.

Failure of an accuracy check will cause the basis to be refactored. Failure of an accuracy check immediately after refactoring will cause the current pivot selection tolerances to be tightened by one increment before another attempt is made. dy_accchk will repeat this cycle until the accuracy checks are satisfied or there's no more room to tighten the pivot selection parameters. On the other hand, each time that an accuracy check is passed without refactoring the basis, the current pivot selection tolerances are loosened by one increment, to a floor given by the minimum pivot selection tolerance.

The minimum pivot selection tolerance is reset to the loosest possible setting at the start of each simplex phase. If groombasis detects and corrects major status errors (indicating that an unacceptable amount of inaccuracy accumulated since the basis was last factored), it will raise the minimum pivot selection tolerance. Similarly, if the primal phase I objective is found to be incorrect, or primal or dual feasibility is lost when attempting to verify an optimal solution, the current and minimum pivot selection tolerances will be raised before returning to simplex pivots. Raising the minimum pivot selection tolerance provides long-term control (for the duration of a simplex phase) over reduction in the current pivot selection tolerance.

The primal accuracy check is $Bx^B = b - Nx^N$. Comparisons are made against the scaled tolerance $\|b\|_1$(dy_tols.pchk). To pass the primal accuracy check, it must be that

$$\|(b - Nx^N) - Bx^B\|_1 \leq \|b\|_1 (\text{dy\_tols.pchk})$$

The dual accuracy check is $yB = c^B$. Comparisons are made against the scaled tolerance $\|c\|_1$(dy_tols.dchk). To pass the dual accuracy check, it must be that

$$\|c^B - yB\|_1 \leq \|c\|_1 (\text{dy\_tols.dchk})$$

The primal feasibility check is $l \leq x \leq u$. For each variable, it must be true that $x_j \geq l_j -$ (dy_tols.pfeas)$(1 + |l_j|)$ and $x_j \leq u_j +$ (dy_tols.pfeas)$(1 + |u_j|)$. In the implementation, only the basic variables are actually tested; nonbasic variables are assumed to be within bound as an invariant property of the simplex algorithm. dy_tols.pfeas is scaled from dy_tols.zero as

$$\text{dy\_tols.pfeas} = \min(1, \log\left(\frac{\|x_B\|_1}{\sqrt{m}}\right))(\text{dy\_tols.zero})(\text{dy\_tols.pfeas\_scale}).$$

The dual feasibility check is $\overline{c} = c^N - yN$ of appropriate sign. For each variable, it must be true that $\overline{c}_j \leq$ dy_tols.dfeas for x_j nonbasic at u_j and $\overline{c}_j \geq -$dy_tols.dfeas for x_j nonbasic at l_j. dy_tols.dfeas is scaled from dy_tols.cost as

$$\text{dy\_tols.dfeas} = \min(1, \log\left(\frac{\|y_k\|_1}{\sqrt{m}}\right))(\text{dy\_tols.cost})(\text{dy\_tols.dfeas\_scale}).$$

9 Scaling

DYLP provides the capability for row and column scaling of the original LP problem. This section develops the algebra used for scaling and unscaling and describes some additional details of the implementation.

Let R be a diagonal matrix used to scale the rows of the LP problem and S be a diagonal matrix used to scale the columns of the LP problem. The original problem (1) is scaled as

$$\begin{aligned} \min\,(cS)(S^{-1}x) \\ (RAS)(S^{-1}x) \leq (Rb) \\ (S^{-1}l) \leq (S^{-1}x) \leq (S^{-1}u) \end{aligned}$$

to produce the scaled problem

$$\begin{aligned} \min \breve{c}\breve{x} \\ \breve{A}\breve{x} \leq \breve{b} \\ \breve{l} \leq \breve{x} \leq \breve{u} \end{aligned}$$

where $\breve{A} = RAS$, $\breve{b} = Rb$, $\breve{c} = cS$, $\breve{l} = S^{-1}l$, $\breve{u} = S^{-1}u$, and $\breve{x} = S^{-1}x$. DYLP then treats the scaled problem as the original problem.

In order to report the solution, DYLP generates unscaled values. Recovering unscaled values of the nonbasic primal variables is trivial — they can be read from the original unscaled l and u vectors. To recover the values of the basic variables, DYLP calculates $x^B = S^B\breve{x}^B$.

To recover the unscaled dual variables y, start with $\breve{y} = \breve{c}^B\breve{B}^{-1}$. Then

$$\begin{aligned} \breve{y} &= \breve{c}^B(RBS^B)^{-1} \\ &= (c^BS^B)((S^B)^{-1}B^{-1}R^{-1}) \\ &= c^BB^{-1}R^{-1} \\ &= yR^{-1} \end{aligned}$$

and $y = \breve{y}R$.

By default, DYLP will calculate scaling matrices R and S and scale the constraint system unless the coefficients satisfy the conditions $.5 < \min_{ij}|a_{ij}|$ and $\max_{ij}|a_{ij}| < 2$. The client can forbid scaling entirely, or supply a pair of vectors that will be used as the diagonal coefficients of R and S.

A few additional details are helpful to understand the implementation. DYLP scales the original constraint system before generating logical variables. Nonetheless, it is desirable to maintain a coefficient of 1.0 for each logical (-1.0 in the case of $\geq$ constraints). The row scaling coefficient r_{ii} for constraint i is already determined. To keep the coefficients of logical variables at ± 1.0, the column scaling factor is chosen to be $1/r_{ii}$ and the column scaling matrix S is extended to include logical variables.

In order to provide a client program with a general ability to price a dual pivot[2] without exporting knowledge of the scaling vectors, DYLP calculates unscaled rows of the basis inverse. Given

[2]One use of this capability is the calculation of standard up and down penalties. Another is estimating the degradation in the objective function after adding a branching hyperplane.

$\breve{B}^{-1} = (S^B)^{-1} B^{-1} R^{-1}$, a row k of the inverse will be

$$\breve{\beta}_k = s_{B(k)} \beta_k R^{-1}$$

and

$$\beta_k = \frac{1}{s_{B(k)}} \breve{\beta}_k R$$

where $B(k)$ represents the index of the variable x_j which is basic in position k of the basis. DYLP provides routines which will price nonbasic variables (dy_pricenbvars) and price a dual pivot (dy_pricedualpiv) using unscaled coefficients.

10 Startup

DYLP provides a cold, warm, and hot start capability. For a cold start, DYLP selects a set of constraints and variables to be the initial active constraint system and then crashes a basis. For a warm start, DYLP expects that the caller will supply a basis but assumes that the active constraint system and other data structures need to be built to this specification. For a hot start, DYLP assumes that its internal data structures are valid except for possible modifications to variable bounds, objective coefficients, or right-hand-side coefficients. It will incorporate these modifications and continue with simplex iterations.

DYLP will default to attempting a hot start unless specifically requested to perform a warm or cold start. For all three start types, DYLP will evaluate the constraint system for primal and dual feasibility, choosing primal simplex unless the constraint system is dual feasible and primal infeasible.

It is not possible to perform efficient and foolproof checks to determine if the client has violated the restrictions imposed for a hot start. At minimum, such a check would require a coefficient by coefficient comparison of the constraint system supplied as a parameter with the copy held by DYLP from the previous call. It is the responsibility of the client to notify DYLP if variable bounds, objective coefficients, or right-hand-side coefficients have been changed. DYLP will scan for changes and update its copy of the constraint system only if the client indicates a change.

Section 16 provides detailed information on the options used to control DYLP's startup actions.

The startup sequence for DYLP is shown in Figure 2. The first actions are determined by the purpose of the call. The call may be solely to free retained data structures; if so, this is done and the call returns. The next action is to determine the type of start — hot, warm, or cold — requested by the client. If a warm or cold start is requested, any state retained from the previous call is useless and all retained data structures are freed. For all three types of start, options and tolerances are updated to reflect the parameters supplied by the client.

For a warm or cold start, the constraint system is examined to see if it should be scaled, and the options specified by the client are examined to see if scaling is permitted. If this assessment determines that scaling is advisable and permitted, the constraint system is scaled as described in §9. The original constraint system is cached and replaced by the scaled copy. In the case of a hot start, the existing scaled copy, if present, is retrieved for use. The original system is not consulted again until the solution is packaged for return to the client.

Following scaling, the active constraint system is constructed for a warm or cold start, or modified for a hot start; §§10.1 – 10.3 describe the actions in detail. At the completion of this activity, the active constraint system is assessed for primal and dual feasibility and an appropriate simplex phase is chosen.

Once the constraint system is constructed, common initialisation actions are performed: Data structures are initialised for PSE and DSE pricing, for the perturbation-based antidegeneracy algorithm, and for the pivot rejection algorithm.

To complete the startup sequence, DYLP evaluates the constraint system and client options to determine if it should perform constraint activation or variable activation or deactivation before starting simplex iterations. Variable deactivation is mutually exclusive to constraint and variable activation; the former is considered only during a cold start, the latter only during a warm or hot start.

An initial round of variable deactivation is performed during a cold start if the number of active variables exceeds the number specified by the coldvars option. This activity is intended to

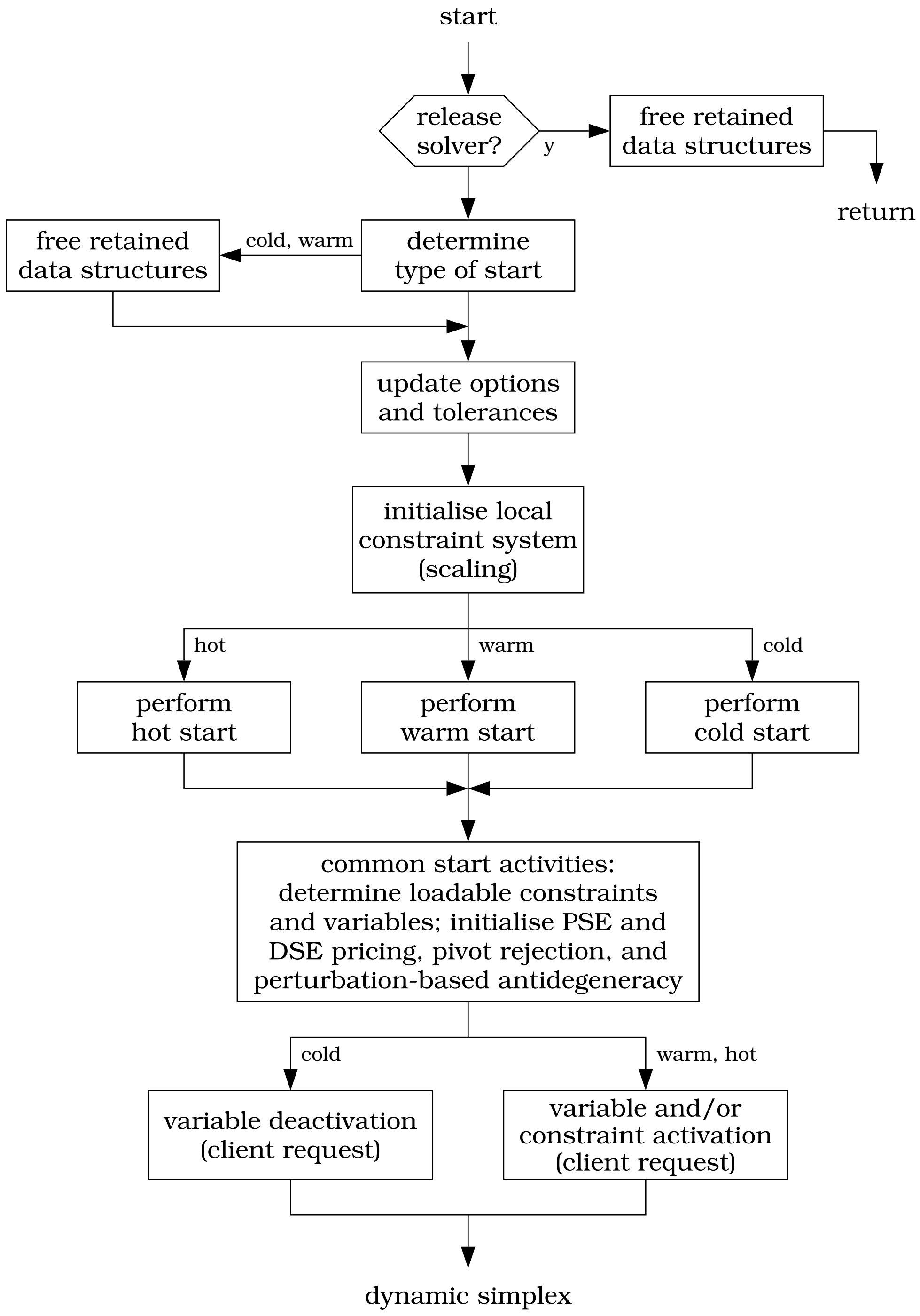

Figure 2: DYLP startup sequence

reduce the initial size of constraint systems with very large numbers of variables (*e.g.*, set covering formulations).

Constraint or variable activation, or both, are performed during a warm or hot start if requested by the client. Constraint activation is performed before variable activation. If initial constraint activation is requested, DYLP will add all violated constraints to the active system. If constraints are added, primal feasibility will be lost, and DYLP will reassess the choice of initial simplex phase.

If initial variable activation is requested, the action taken depends on the initial simplex phase. If DYLP will enter primal simplex, variables with favourable primal reduced costs are activated, evaluated under the phase I or phase II objective as appropriate. For dual simplex, variables which will tend to bound the dual problem are selected for activation: For each infeasible primal basic variable (nonbasic dual variable with favourable reduced cost), primal variables with optimal reduced costs (feasible dual constraints) which will bound motion in the direction of the incoming dual variable are selected for activation.

10.1 Cold Start

DYLP performs a cold start in two phases. The first phase, implemented in dy_coldstart, constructs the initial active constraint system. The second phase, implemented in dy_crash, constructs the initial basis.

To construct the initial active constraint system, dy_coldstart first checks to see if the client has specified that the full constraint system should be used. In this case, the active system will be the entire constraint system and the dynamic simplex algorithm will reduce to a single execution of either primal or dual simplex.

If the client specifies that DYLP should work with a partial constraint system, the constraints are first separated into equalities and inequalities. All equalities are included in the initial active system.

The remaining inequalities are sorted, using the angle of the constraint normal a_i to the objective function normal c as the figure of merit,

$$a_i \angle c = \frac{180}{\pi} \cos^{-1} \frac{a_i \cdot c}{\|a_i\| \, \|c\|}$$

Consider a minimisation objective and '≤' inequalities. The normals of the inequalities point out of the feasible region, and the normal of the objective function will point into the feasible region at optimality. Hence a constraint whose normal forms an angle near 180° with the normal of the objective should be more likely to be active at optimum. A constraint whose normal forms an angle near 0° is more likely to define a facet on the far side of the polytope. Unfortunately, 'more likely' is not certainty, and it's easy to construct simple two-dimensional examples where the normal of one of the constraints active at optimality forms an acute angle with the normal of the objective function.

DYLP allows the client to specify one or two angular intervals and a sampling fraction which are used to select inequalities to add to the initial active system. By default, the initial system will be populated with 50% of the inequalities which form angles in the intervals $[0°, 90°)$ and $(90°, 180°]$. (*I.e.*, inequalities whose normals are perpendicular to the objective normal are excluded entirely, and half of all other inequalities will be added to the initial active system.) The inequalities selected will be spread evenly across the specified range(s). DYLP will activate all variables referenced by each constraint.

Once the initial constraint system is populated, dy_crash is called to select an initial basis. DYLP offers three options for the initial basis, called 'logical', 'slack', and 'architectural'. A logical basis is the standard unit basis composed of slack and artificial (logical) variables for the active constraints. A slack basis again uses slack variables for inequalities, but attempts to select architectural variables for equalities, including artificial variables only if necessary. An architectural basis attempts to choose architectural variables for all constraints, selecting slack and artificial variables only when necessary.

There are many qualities which are desirable in an initial basis, and they are often in conflict. A logical basis is trivially easily to construct, factor, and invert, and has excellent numerical stability. On the other hand, such a basis is hardly likely to be the optimal basis. When choosing architectural variables, free variables are highly desirable since they will never leave the basis. In addition, DYLP's basis construction algorithm tries to select architectural variables which will form an approximately lower-diagonal matrix and provide numerically stable pivots. Constructing a matrix which is approximately lower-diagonal minimises fill-in when the basis is factored. Several of the ideas implemented in DYLP's initial basis construction algorithms are described by Bixby in [1].

Since DYLP makes an effort to populate the constraint system with constraints that should be tight at optimality, an architectural basis is the default.

10.2 Warm Start

The routine dy_warmstart implements a warm start. The client is expected to supply an initial basis, expressed as a set of active constraints and corresponding basic variables. By default, DYLP will activate all variables referenced by each constraint. As an option, the client can specify an initial set of active variables.

10.3 Hot Start

For a hot start, DYLP assumes that all internal data structures are exactly as they were when it last returned to the client. Changes to the constraint system must be confined to the right-hand-side, objective, and variable upper and lower bound vectors, so that the basis factorisation and inverse are not affected. The client is responsible for indicating to DYLP which of these vectors have been changed. The routine dy_hotstart scans the changed vectors and orchestrates any updates to the corresponding data structures in the active constraint system. Unlike a cold or warm start, the basis is *not* factored prior to resuming pivots. DYLP assumes that the basis was refactored as part of the normal preoptimality sequence prior to the last return to the client and that no intervening pivots have occurred. Any numerical problems arising from the modifications specified by the client will be picked up in the normal course of dynamic simplex execution.

11 Dynamic Simplex

11.1 Normal Algorithm Flow

Figure 3 gives the normal flow of the dynamic simplex algorithm implemented in DYLP. The outcomes included in the normal flow of the algorithm are primal optimality, infeasibility, and unboundedness, and dual optimality and unboundedness. Other outcomes (*e.g.*, loss of dual feasibility during dual simplex, or numerical instability) are discussed in §11.2.

The implementation of the dynamic simplex algorithm is structured as a finite state machine, with six normal states, primal simplex, dual simplex, deactivate variables, activate variables, deactivate constraints, and activate constraints; two user-supplied states, generate variables and generate constraints; and three error recovery states, force primal feasibility, force dual feasibility, and force full constraint system. State transitions are determined by the previous state, the type of simplex in use, and the outcome of actions in a state.

As described in §10, DYLP establishes an initial active constraint system, determines whether the system is primal or dual feasible, and chooses the appropriate simplex as the starting phase.

The most common execution pattern is as described in the Introduction: The initial active constraint system is neither primal or dual feasible. Primal simplex is used to solve this system to optimality. A minor loop then activates variables with favourable reduced cost and reoptimises using primal phase II. This loop repeats until no variables can be activated; at this point the solution is optimal for the active constraints, over all variables. The algorithm then attempts to activate violated constraints; if none are found, the solution is optimal for the original problem. After violated constraints are activated, loose constraints are deactivated and dual simplex is used to reoptimise. When an optimal solution is reached, the algorithm attempts to activate variables with favourable reduced cost and return to the 'primal phase II – activate variables' minor loop. If no variables can be activated, the algorithm attempts to activate violated constraints. If none are found, the solution is optimal for the original problem. If violated constraints are activated, then an attempt is made to activate dual feasible variables and dual simplex is used to reoptimise.

There is an obvious asymmetry in the use of primal and dual simplex. When primal simplex reaches an optimal solution, the 'primal phase II – activate variables' minor loop iterates until no useful variables remain to be activated. Only then does the algorithm activate violated constraints and move to dual simplex. The analogous minor loop for dual simplex would be to add violated constraints (dual variables with favourable reduced costs) and reoptimise with dual simplex until no violated constraints remain. Instead, the algorithm attempts to add variables and return to primal simplex; failing that, it will add both violated constraints and dual feasible variables (satisfied dual constraints). The purpose of this asymmetry is two-fold: It acknowledges that primal infeasibility is much more likely than primal unboundedness when solving LPs in the context of a branch-and-cut algorithm, and it attempts to avoid the large swings in the values of primal variables which often accompany dual unboundedness. Dual simplex moves between primal infeasible basic solutions which can be at a large distance from the primal feasible region and at a large distance from one another in the primal space. This presents a challenge for numerical stability. Because the primal simplex remains within the primal feasible region, primal unboundedness does not present the same difficulty.

To avoid cycling by repeatedly deactivating and reactivating the same constraint when the dimension of the optimal face is greater than one, constraint deactivation is skipped unless there has been an improvement in the objective function since the previous constraint deactivation phase. This guarantees that the simplex will not return to a previous extreme point.

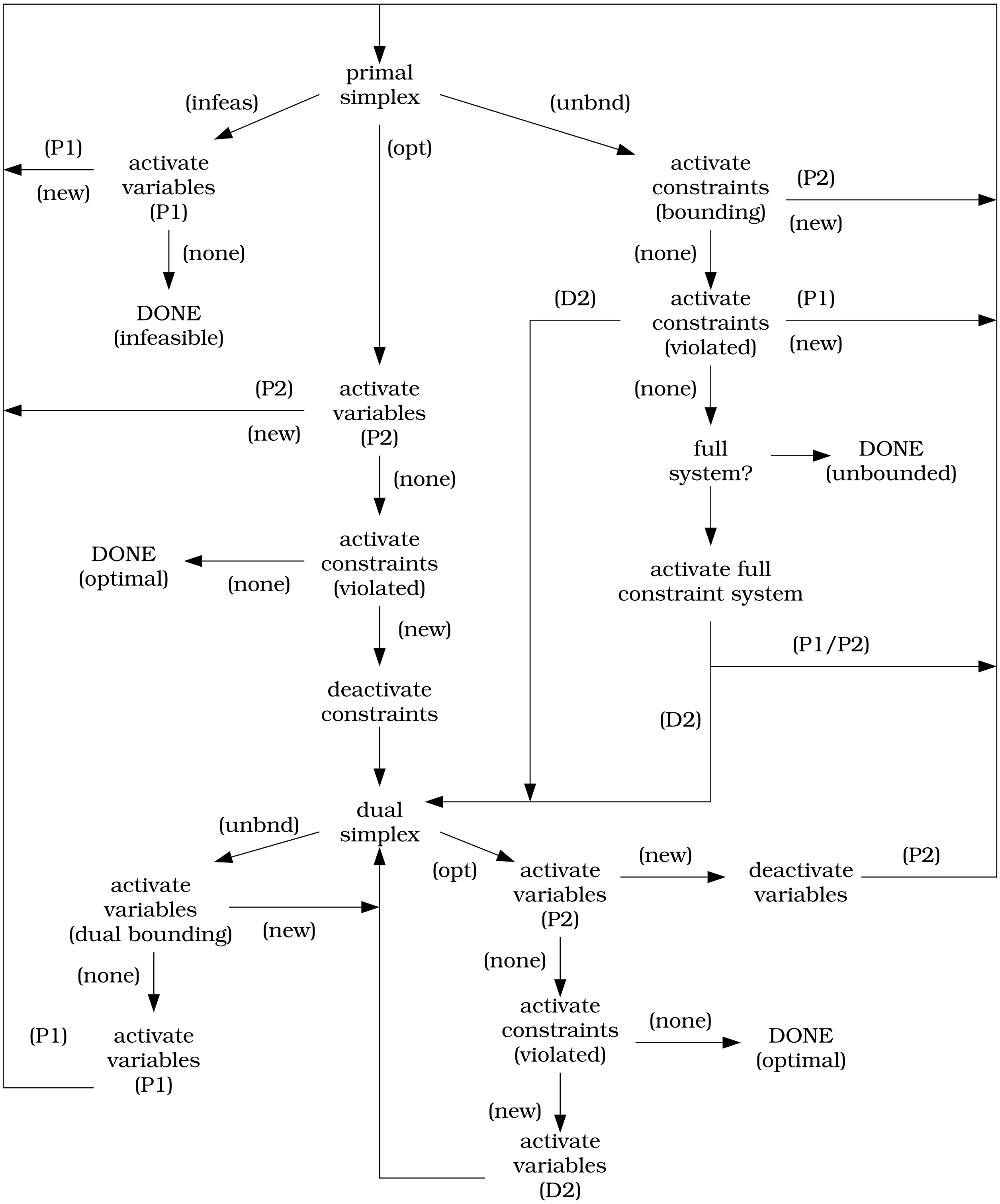

Figure 3: Dynamic Simplex Algorithm Flow

If primal simplex finds that the active system is infeasible, the algorithm will attempt to activate variables with favourable reduced cost under the phase I objective function (*vid.* §13) and resume primal phase I. If no variables can be found, the original problem is infeasible.

If primal simplex finds that the active system is unbounded, the algorithm first attempts to activate bounding constraints which will not cause the loss of primal feasibility. If such constraints can be found, execution returns to primal phase II. If no such constraints can be found, or primal feasibility is not an issue, all violated constraints are added and execution moves to dual simplex. If no violated constraints can be found, the full constraint system is activated. If primal simplex again returns an indication of unboundedness, the original problem is declared to be unbounded. The effort expended before indicating a problem is unbounded acknowledges that unboundedness is expected to be extremely rare in DYLP's intended application.

If dual simplex finds that the active system is dual unbounded (primal infeasible), the algorithm first attempts to activate dual bounding constraints (primal variables) which will not cause the loss of dual feasibility. If such dual constraints can be found, execution returns to dual simplex. If no such dual constraints can be found, the algorithm will attempt to activate variables with favourable reduced cost under the primal phase I objective function and continue with primal phase I.

11.2 Error Recovery

A substantial amount of DYLP's error recovery capability is hidden within the primal and dual simplex algorithms. It is also possible to use the capabilities present in a dynamic simplex algorithm to attempt error recovery at this level. The dynamic simplex algorithm modifies the constraint system as part of its normal execution. This ability can be harnessed to force a transition from one simplex to another when one simplex runs into trouble. The actions described in this section are fully integrated with the actions described in §11.1. They are described separately to avoid reducing Figure 3 to an incomprehensible snarl of state transitions.

Primal Simplex

The error recovery actions associated with the primal simplex algorithm are shown in Figure 4. There are five conditions of interest, excessive change in the value of primal variables (excessive swing), stalling (stall), inability to perform a pivot (punt), numerical instability (accuracy check), and other errors (other error).

Excessive change ('swing') in the value of a primal variable during primal simplex is taken as an indication that the primal problem is verging on unboundedness. Swing is defined as (new value)/(old value). DYLP's default tolerance for this ratio is 10^{15}. The action taken is the same as that used for normal detection of unboundedness, with the exception that the algorithm will always return to primal simplex.

When primal simplex stalls or is forced to punt, the strategy is to attempt to modify the constraint system so that the simplex algorithm will be able to choose a new pivot and again make progress toward one of the standard outcomes of optimality, infeasibility, or unboundedness. The specific actions vary slightly depending on whether primal feasibility has been achieved.

If primal simplex is still in phase I, the first action is to try to activate variables which have a favourable reduced cost under the phase I objective. If this succeeds, execution returns to primal simplex. If no variables can be found, the algorithm will attempt to activate violated constraints; if successful, execution returns to primal simplex. If no variables or constraints have been activated, there is no point in returning to primal simplex as the outcome will be unchanged. In this case,

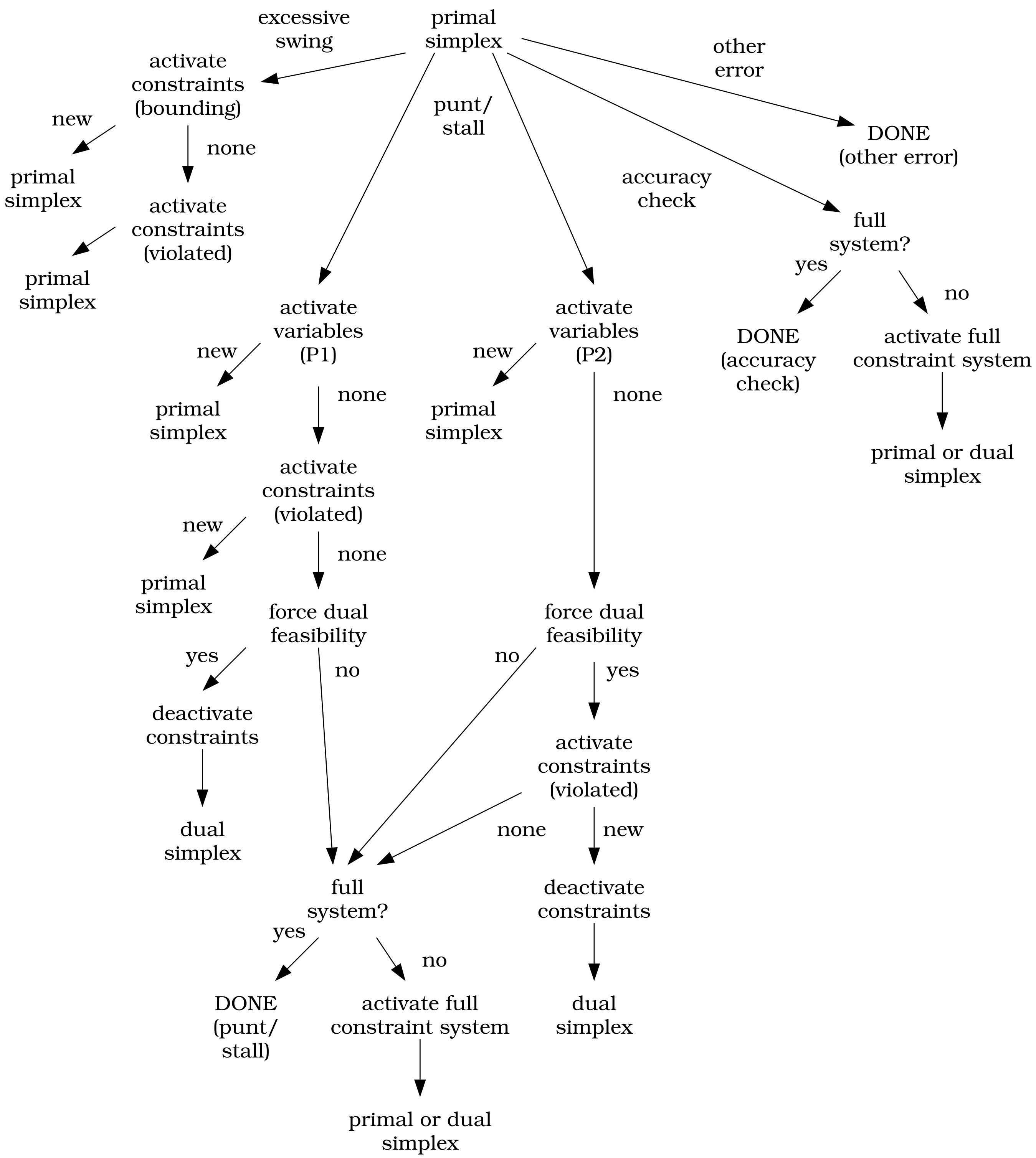

Figure 4: Error Recovery Actions for Primal Simplex Error Outcomes

the algorithm will attempt to force dual feasibility by deactivating variables whose reduced costs are not dual feasible (*i.e.*, deactivate unsatisfied dual constraints). If this succeeds, the algorithm will deactivate loose constraints (dual variables) to reduce the chance of dual unboundedness and continue with dual simplex. Failing all the above, the ultimate action is to active the full constraint system and attempt to solve it with primal or dual simplex. This can be done only once, to avoid a cycle in which the full system is activated, pared down while forcing primal or dual feasibility, and then reactivated when lesser measures again fail.

When a stall or punt occurs in primal phase II, the first action is again to attempt to activate variables with a favourable reduced cost. However, if no new variables can be found, the algorithm immediately attempts to force dual feasibility. Only if this can be achieved will it proceed to activate violated constraints, deactivate loose constraints, and proceed to dual simplex. Failure to force dual feasibility or to activate any constraints causes forced activation of the full constraint system as described above.

Both the primal and dual simplex algorithm incorporate extensive checks and error recovery actions to detect and recover from numerical instability. By the time a simplex gives up and reports that it cannot overcome numerical problems, there is little to be done but force activation of the full constraint system for one last attempt.

Other errors indicate algorithmic failures within the simplex algorithms (*e.g.*, failure to acquire resources, or conditions not anticipated by the code) and no attempt is made to recover at the dynamic simplex level.

Dual Simplex

The error recovery actions associated with the dual simplex algorithm are shown in Figure 5. In addition to the five outcomes cited for primal simplex, loss of dual feasibility (lost dual feasibility) can be reported by the dual simplex algorithm. (Loss of primal feasibility is handled internally by the primal simplex, which simply returns to phase I simplex iterations.)

When the dual simplex algorithm loses feasibility, the algorithm will attempt to force dual feasibility by deleting the offending dual constraints (primal variables). If this succeeds, it will attempt to activate feasible dual constraints and return to dual simplex. If dual feasibility cannot be restored, the algorithm attempts to activate variables with favourable reduced costs under the primal phase I objective and executes primal phase 1.

Excessive change in the value of primal variables during dual simplex is taken as an indication that the dual algorithm is moving between basic solutions which are far outside the primal feasible region and far from each other. When excessive change in a primal variable is detected, the algorithm attempts to activate primal constraints which will bound this motion. If this is successful, execution of dual simplex resumes. General activation of violated primal constraints is not attempted as it is less likely to bound the primal swing. If no bounding constraints can be found, the algorithm attempts to activate feasible dual constraints and return to dual simplex. If no such constraints can be found, the algorithm attempts to activate variables with favourable reduced costs under the primal phase I objective and executes primal phase 1.

When dual simplex reports that it has stalled or cannot execute necessary pivots, the algorithm first attempts to activate violated primal constraints. If such constraints can be activated, execution returns to dual simplex. If no constraints can be found, the algorithm attempts to force primal feasibility by deactivating violated primal constraints. Depending on the result of this action, the algorithm attempts to activate variables with favourable reduced costs under the primal phase I or phase II objective and executes primal simplex.

Loss of numerical stability and other errors are handled as for primal simplex.

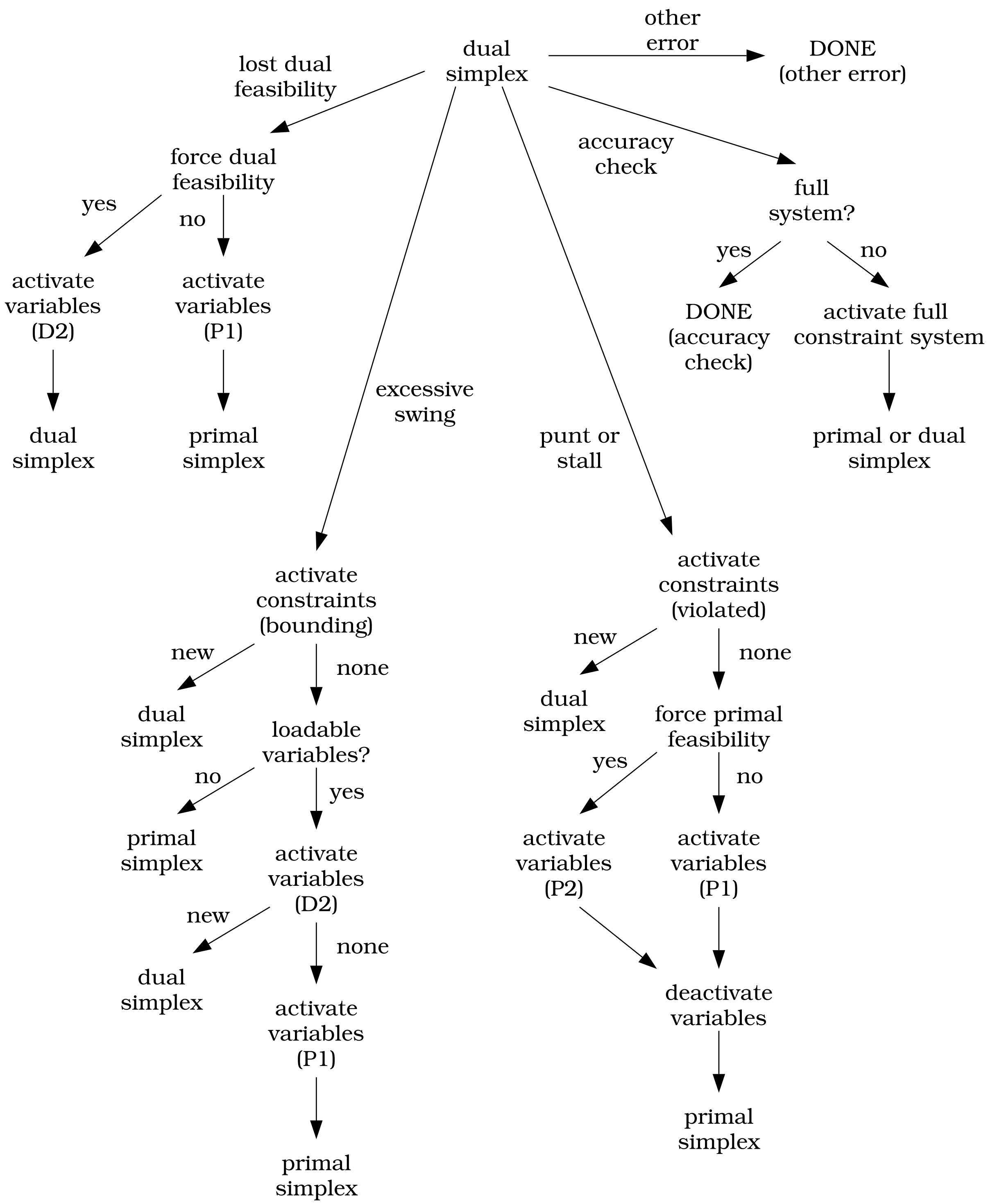

Figure 5: Error Recovery Actions for Dual Simplex Error Outcomes

12 Dual Simplex

DYLP will choose dual simplex whenever the current basic solution is dual feasible but not primal feasible. The primary role of dual simplex in DYLP is reoptimisation following the addition of violated constraints. The implementation reflects this role and does not provide a dual phase I for achieving dual feasibility. The dual simplex implementation incorporates dual steepest edge (DSE) pricing (§4.2), standard (§12.5) and generalised (§12.6) pivoting, and perturbation-based (§5) and alignment-based (§6) antidegeneracy algorithms.

Because the dual simplex implementation does not provide a phase I, a number of exceptional conditions will cause DYLP fall back from dual simplex to primal simplex.

In dynamic simplex, apparent primal infeasibility can result because only a subset of the variables are present in the active constraint system. In some cases, the variables needed to regain feasibility cannot be activated into the nonbasic partition while maintaining dual feasibility. In the context of the dual problem, the problem is unbounded and any dual constraint which would bound it would also make the current basic solution dual infeasible. DYLP implements a variable activation procedure which can pivot a single variable into the basis as it is activated in order to maintain dual feasibilty. It is still possible, however, to reach a basic solution where multiple pivots are required to regain dual feasibility for any candidate variable. When this occurs, DYLP reverts to primal simplex.

If primal infeasible variables remain but they cannot be pivoted because their pivot coefficients do not satisfy the current pivot selection tolerances, dy_dual will punt and DYLP will return to phase I of the primal simplex algorithm in the hope that addition of variables and/or the application of primal pivoting rules will allow pivoting to continue. In addition, if the dual simplex terminates due to stalling or loss of feasibility, DYLP will try the primal simplex algorithm before giving up.

Figure 6 shows the call structure of the dual simplex implementation.

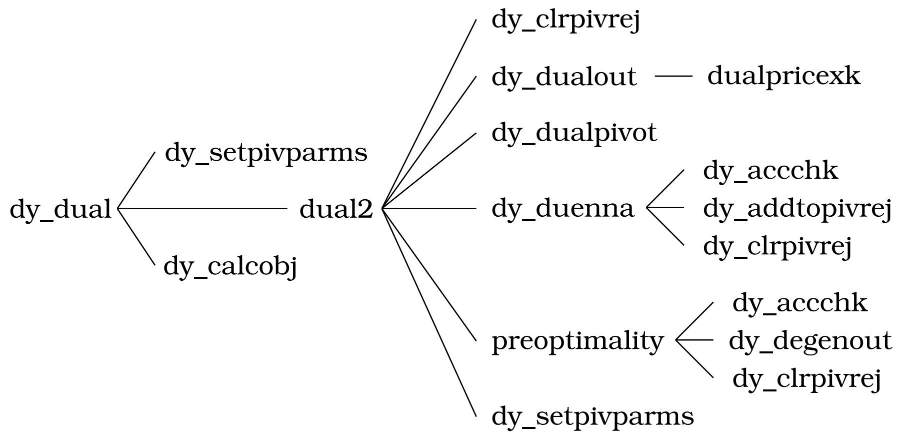

Figure 6: Call Graph for Dual Simplex

12.1 Dual Top Level

Dual simplex is executed when the dynamic simplex state machine enters state dyDUAL. If required, DSE pricing is initialised by calculating the square of the norms of the rows of the basis inverse (*vid.* §4.2) and the dual simplex routine dy_dual is called. dy_dual is a trivial shell which calculates the objective (dy_calcobj) and calls the dual phase II routine dual2 to do the optimisation.

12.2 Dual Phase II

The overall flow of phase II of the dual algorithm is shown in Figure 7. The body of the routine is structured as two nested loops. The outer loop handles startup and termination, and the inner loop handles the majority of routine pivots.

On entry to dual2, the outer loop is entered and dy_dualout is called to select the initial leaving variable. Then the inner loop is entered and dy_dualpivot is called to perform the pivot. dy_dualpivot (*vid.* §12.3) will calculate the coefficients of the pivot row (dualpivrow), select an entering variable (dualin), pivot the basis (dy_pivot), update the primal and dual variables (dualupdate), and update the DSE pricing information and reduced costs (dseupdate). For a routine pivot, dseupdate will also select a leaving variable for the next pivot. dy_duenna evaluates the outcome of the pivot, handles error detection and recovery where possible, and performs the routine maintenance activities of accuracy checks and refactoring of the basis. If there are no problems, the pivoting loop iterates, using the leaving variable selected in dseupdate. The loop continues until optimality is reached, the problem is determined to be primal infeasible (dual unbounded), or an exception or fatal error occurs.

One common reason for a failure to select a leaving variable for the next pivot is that the current pivot was aborted due to numerical problems (an unsuitable pivot coefficient being the most common of these). In this case, dseupdate never executes. Once dy_duenna has taken the necessary corrective action, the flow of control escapes to the outer loop and calls dy_dualout to select a new leaving variable.

Another common reason for failure to select a leaving variable is that all candidates were previously flagged as unsuitable pivots. In this case, dy_dualout will indicate a 'punt' and dy_dealWithPunt will be called to reevaluate the flagged variables. If it is able to make new candidates available, control returns to dy_dualout for another attempt to find a leaving variable. If all flagged variables remain unsuitable, control flow moves to the preoptimality actions with an indication that dual simplex has punted.

When dy_dualout indicates optimality (primal feasibility) or dy_dualpivot indicates optimality, dual unboundedness (primal infeasibility), or loss of dual feasibility, the inner loop ends and preoptimality is called for confirmation. preoptimality will refactor the basis, check for accuracy, recompute the primal and dual variables, and confirm dual and primal feasibility status. If there are no surprises, dual phase II terminates with an indication of optimality, dual unboundedness, or loss of dual feasibility.

Loss of dual feasibility stems from loss of numeric accuracy, but it cannot be corrected within dual phase II. The error recovery actions taken by the dynamic simplex algorithm are described in §11.2.

Loss of primal feasibility can occur for two distinct reasons. In the less common case, loss of primal feasibility stems from loss of numeric accuracy. The pivot selection rules are tightened and

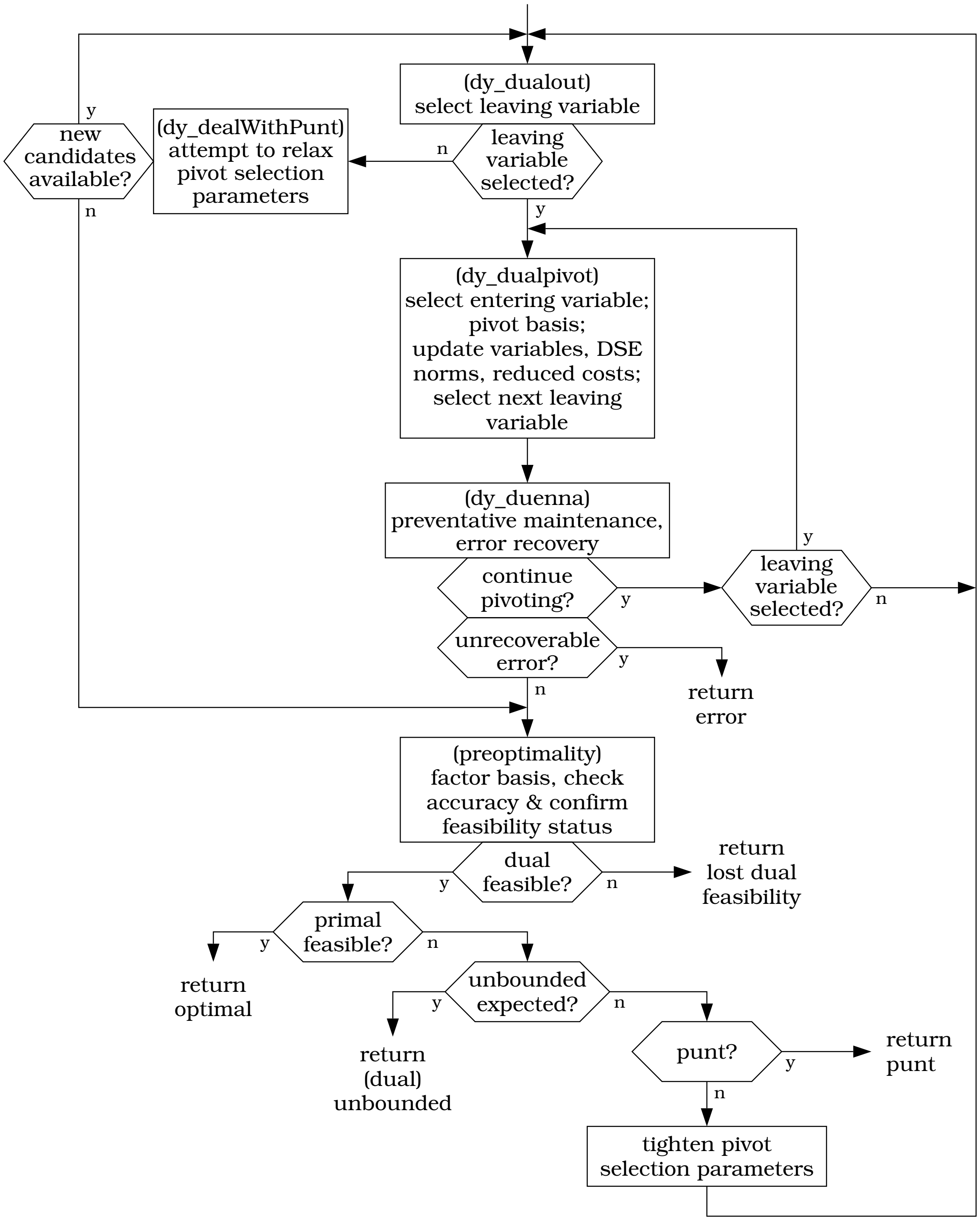

Figure 7: Dual Phase II Algorithm Flow

dual simplex iterations are resumed. When the number of false indications of optimality exceeds a hard-coded limit (currently 15), dual simplex terminates with a fatal error.

The more common reason for apparent loss of primal feasibility at the termination of dual simplex is that it is ending with a punt, as described above. The variables flagged as unsuitable for pivoting are not primal feasible, and when the flags are removed to perform the preoptimality checks, primal feasibility is revealed as an illusion. No further action is possible within dual simplex; the reader is again referred to §11.2.

Other errors (*e.g.*, stalling, accuracy checks, *etc.*) not shown in Figure 7 can occur and result in termination of the dual simplex algorithm with the appropriate error indication.

12.3 Pivoting

DYLP offers two flavours of dual pivoting: A standard dual pivot algorithm in which a single primal variable is selected and pivoted into the basis, and a generalised dual pivot algorithm [8, §10.2] in which multiple primal variables may undergo bound-to-bound flips prior to the basis pivot. The choice of standard or generalised dual pivoting can be controlled with an option; DYLP will use generalised pivoting by default.

Figure 8 shows the call structure of the dual pivot algorithm. The routine dualin implements standard dual pivoting; dualmultin implements generalised dual pivoting.

The first activity in dy_dualpivot is the calculation of the coefficients of the pivot row, $\overline{a}_i = \beta_i N$, by the routine dualpivrow. With the leaving primal variable and the basis inverse row in hand, one of dy_dualin or dualmultiin are called to select the entering variable. (If generalised dual pivoting is in use, dualmultiin will perform any bound-to-bound flips before returning.)

Once the entering and leaving variables have been chosen, the actual pivot is performed in several steps. Prior to the pivot, the vector $\tau = B^{-1}\beta_k^T$ is calculated for use during the update of the DSE pricing information. The basis is pivoted next; this involves calls to dy_ftran and dy_pivot, as outlined in §7.3. If the basis change succeeds, the primal and dual variables are updated by dualupdate using the iterative update formulæ of §3, and then the DSE pricing information and reduced costs are updated by dseupdate, using the update formulæ of §4.2. As a side effect, dseupdate will select a leaving variable for the next pivot.

12.4 Selection of the Leaving Variable

The selection of the leaving primal variable x_i (entering dual variable y_i) is made using the dual steepest edge criterion described in §4.2. As outlined above, the normal case is that the leaving variable for the following pivot will be selected as dseupdate updates the DSE pricing information for the current pivot. In various exceptional circumstances where this does not occur, the routine dy_dualout is called to make the selection.

12.5 Standard Dual Pivot

For the standard dual pivot algorithm, the selection of the entering primal variable (leaving dual variable) is made using the usual dual pivoting rules and a set of tie-breaking strategies.

Let x_i be the leaving primal variable, for simplicity of exposition occupying basis position i. β_i is obtained by calling dy_btran to calculate $e_i B^{-1}$, where e_i is the unit row vector with 1 in position i.

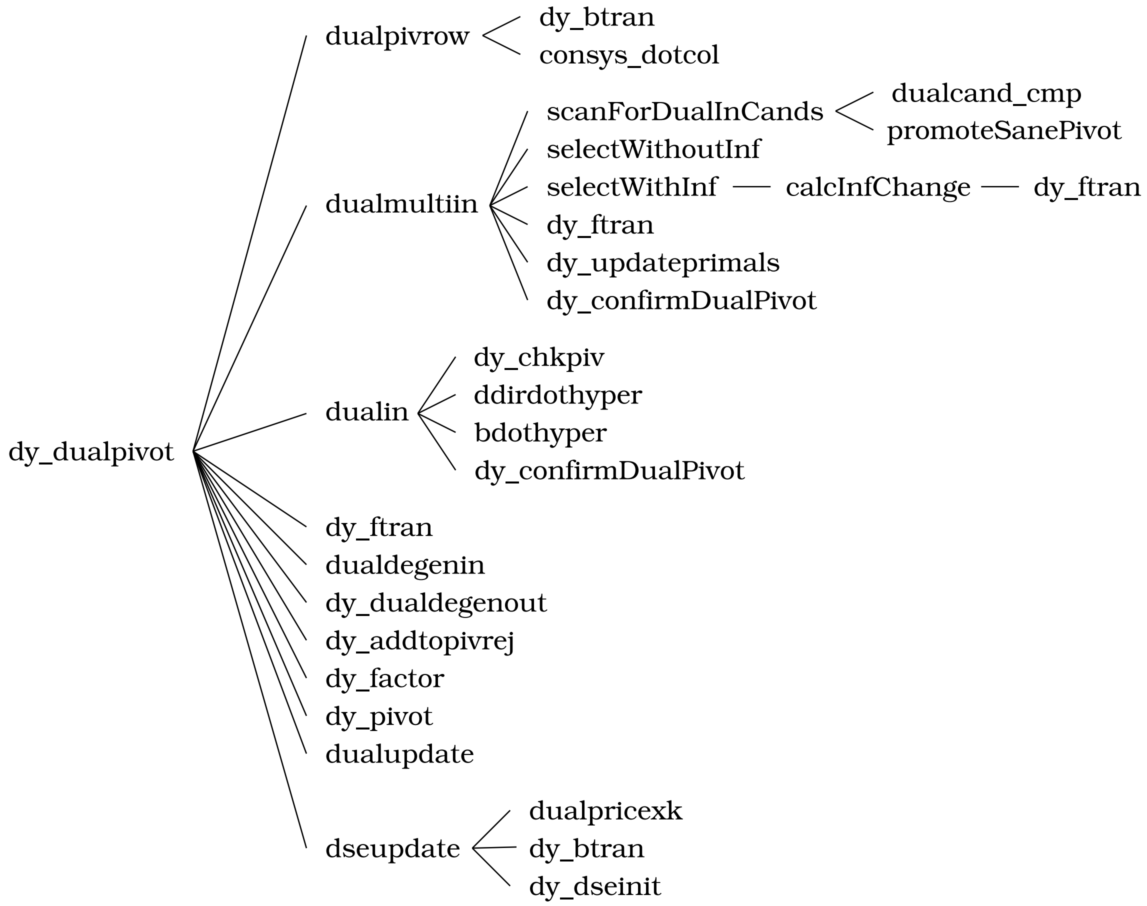

Figure 8: Call Graph for Dual Pivoting

The pivot coefficient for a variable x_k is $\overline{a}_{ik} = \beta_i a_k$. Let y_i be the dual variable associated with the constraint in basis position i and let y_k be the dual variable associated with the tight bound constraint for the nonbasic primal variable x_k.

Abstractly, we need to check $y_k = \overline{c}_k + \overline{a}_{ik}\delta_{ik}$ to find the maximum allowable δ_{ij} such that $y_k \geq 0 \; \forall k \in \mathcal{B}$ and $y_j = 0$ for some j. The index j of the entering primal variable x_j will be

$$j = \arg\min_{k} \left| \frac{\overline{c}_k}{\overline{a}_{ik}} \right| \tag{13}$$

for suitable $x_k \in N$.

In practice, it's impossible to explain 'suitable x_k' properly without going deep into the details of the workings of the revised dual simplex algorithm (*vid.* [4]). Table 1 gives the rules in tabular form, from the perspective that when all is said and done, the leaving primal variable must end up nonbasic at bound and the sign of the reduced cost must be appropriate for that bound. Interpreting the table, the second line says that if the leaving variable will be made primal feasible by rising to its lower bound, the resulting reduced cost must be positive to retain primal optimality, hence the corresponding dual variable must enter by rising from zero. If the entering primal variable will be decreasing from its upper bound, the current reduced cost must be negative,

leaving x_i	entering y_i	resulting $\overline{c}_i$	entering x_j	leaving y_j	initial $\overline{c}_j$	pivot $\overline{a}_{ij}$	$-\dfrac{\overline{c}_j}{\overline{a}_{ij}} = \overline{c}_i$
$\nearrow lb$	$0 \nearrow$	≥ 0	$lb \nearrow$	$\searrow 0$	≥ 0	< 0	$-\dfrac{(+)}{(-)} = (+)$
			$ub \searrow$	$\nearrow 0$	≤ 0	> 0	$-\dfrac{(-)}{(+)} = (+)$
$\searrow ub$	$0 \searrow$	≤ 0	$lb \nearrow$	$\searrow 0$	≥ 0	> 0	$-\dfrac{(+)}{(+)} = (-)$
			$ub \searrow$	$\nearrow 0$	≤ 0	< 0	$-\dfrac{(-)}{(-)} = (-)$

Table 1: Summary of Dual Simplex Pivoting Rules

hence the corresponding dual variable must leave by rising to zero[3]. The final columns simply illustrate that the sign of the pivot is well-defined from the update formula.

DYLP provides a selection of tie-breaking strategies when there are multiple candidates with equal $|\delta_{ik}| = \delta_{\min}$. The simplest is to select the first variable x_k such that $\delta_{ik} = 0$. A slightly more sophisticated strategy is to scan all variables x_k eligible to enter and pick x_j such that $j = \arg\max_{k \in K} |\overline{a}_{ik}|$, $K = \{k \mid |\delta_{ik}| = \delta_{\min}\}$; DYLP will use this strategy by default. DYLP also provides four additional strategies based on hyperplane alignment as described in §6. An option allows the tie-breaking strategy to be selected by the client.

In case of degeneracy, the perturbed subproblem anti-degeneracy algorithm described in §5 is also available. The client can control the use of perturbed subproblems through two options which specify whether a perturbed subproblem can be used, and how many consecutive degenerate pivots must occur before the perturbed subproblem is created. By default, DYLP uses perturbed subproblems aggressively and will introduce one when faced with a second consecutive degenerate pivot.

12.6 Generalised Dual Pivot

Suppose that an entering dual variable y_i has been chosen, and the ratio test of equation (13) has been used to select a leaving variable y_j and determine the change δ_{ij} in y_i required to drive $y_j = \overline{c}_j$ to zero. Generalised dual pivoting asks the question "What happens when we push past this limit?"

Immediately, dual feasibility is lost as the value of y_j changes sign. But ... suppose that the corresponding nonbasic primal variable x_j has both an upper and lower bound. If the value of this variable is changed to the opposite bound ('flipped'), the sign of y_j is again correct with respect to the value of x_j and dual feasibility is restored. Flipping x_j will change the value of any basic primal variable x_k where $\overline{a}_{kj} = \beta_k a_j \neq 0$. In particular, the value of x_i will move toward feasibility. In terms of dual simplex, the reduced cost $\overline{b}_i = x_i$ of y_i will be reduced. If $\overline{b}_i$ is not yet reduced to zero, y_i can still be used as the entering dual variable (albeit with a less favourable reduced cost)

[3] Properly accounting for these *apparently* negative dual variables is the difficulty in trying to explain pivoting from the dual simplex perspective. In fact, negative dual variables are an artifact of running the dual simplex algorithm using representation and data structures appropriate for primal simplex with implicit bound constraints.

and the ratio test can be repeated to determine a new leaving dual variable $y_{f'}$. Repeating this procedure will identify a maximum sequence of primal variable flips. The sequence ends for one of two reasons:

- The primal variable x_f associated with a dual variable y_f has only one finite bound and cannot be flipped.
- Flipping the primal variable x_f will push x_i over its bound and into feasibility. In dual simplex terms, y_i will acquire an unfavourable reduced cost and will no longer be a suitable choice for the entering dual variable.

The dual variable y_f corresponding to x_f becomes the leaving dual variable. The dual basis pivot will have y_f leaving and y_i entering; the corresponding primal pivot has x_i leaving and x_f entering. This sequence of primal variable flips culminating in a final pivot is generalised dual pivoting. Note that it's possible to choose any variable within the maximum sequence of flips and use it as the pivot variable.

DYLP implements generalised dual pivoting by first collecting the set of potential leaving dual variables y_k (and associated entering primal variables x_k). This set is then sorted using non-decreasing value of $|\delta_{ik}|$ and numerical stability of the pivot as the primary and secondary sort criteria. (Numerical stability is a binary condition for this purpose; a pivot is either acceptable or not.) The tertiary sort criterion varies according to whether $\delta_{ik} = 0$ or $\delta_{ik} \neq 0$.

- For variables with $\delta_{ik} = 0$, give preference to primal variables which can be flipped to their opposite bound.
- For variables with $\delta_{ik} \neq 0$, give preference to variables which cannot be flipped.

Any remaining ties are broken with a preference for pivot coefficients with better numerical stability (compared as an analog value). This final tie-breaking criterion is important when flipping a sequence of variables because numerical stability is relative to the largest coefficient value $|\overline{a}_{iq}| = \max_k |\overline{a}_{ik}|$ in a column. An unstable pivot has a small ratio $|\overline{a}_{ik}/\overline{a}_{iq}|$; this implies a high probability that when x_k is flipped, other basic primal variables (at the least, x_q) will incur large changes. Stability of primal variable values is thus improved by preferring large pivot coefficients.

A nondegenerate dual pivot is clearly preferable to a degenerate pivot, and this motivates the preference for flippable variables within the set of candidates with $\delta_{ik} = 0$. Ideally, all variables in this group can be flipped; failing this, it's preferable to flip as many as possible. When consideration moves into the group of candidates with $\delta_{ik} \neq 0$, the goal changes. Quick selection of a good pivot will minimise further unpredictable changes to other dual reduced costs (primal basic variables). Since pivoting is the goal, it is reasonable to give preference to variables that must be pivoted.

The process of scanning for candidates and sorting the resulting set is implemented in the routines scanForDualInCands and dualcand_cmp.

The sorting procedure just described may result in an ordered list where one or more unflippable candidates y_u with numerically unstable pivots $\overline{a}_{iu}$ precede the first candidate y_s with a stable pivot $\overline{a}_{is}$. In this case, a final attempt is made to promote the candidate with a stable pivot so that it precedes the the unsuitable candidates y_u. From the sort criteria, it must be that $|\delta_{is}| \geq |\delta_{iu}|$. For a given variable y_u, if $|y_u - \overline{a}_{iu}\delta_{is}|$ is less than the dual feasibilty tolerance, the resulting dual infeasibility will be tolerable and y_s can be promoted over y_u. This promotion of a stable pivot over an unstable pivot is implemented in promoteSanePivot.

At the end of the above sort algorithm, the list of candidates is ordered so that it begins with a maximum sequence of flippable variables, followed by a variable which must be pivoted. The routine selectWithoutInf scans the sorted list and selects the actual pivot variable according to the criteria specified above for a maximum sequence of flips and final pivot.

DYLP implements one additional experimental capability within generalised dual pivoting. As mentioned above, flipping nonbasic primal variables will, in general, change the values of an arbitrary set of the basic primal variables. It is possible, but expensive, to track this change; the major cost is the calculation of $\overline{a}_k = B^{-1} a_k$ for each candidate column. With this information in hand, it is possible to locate, within the sequence of variables eligible to be flipped or pivoted, the point at which the maximum primal infeasibility is at a minimum over the basic variables; this variable becomes the pivot variable. This method of selecting the pivot variable is implemented in the routine selectWithInf.

Computational experience shows that using the minimum maximum primal infeasibility to choose the pivot variable x_f is not a good strategy when dual simplex is behaving well. Dual simplex moves through a sequence of primal infeasible basic solutions. Observation of dual simplex in operation often shows a pattern where the values of primal variables grow increasingly infeasible and then, within a relatively few pivots, collapse to feasibility (hence optimality). Attempting to suppress the initial growth of primal infeasibility is counterproductive, lengthening the sequence of pivots required to attain optimality. However, very large infeasible primal values present challenges to numerical accuracy, so that it may be desirable in extreme cases to choose pivots with a goal of reducing primal infeasibility.

DYLP by default implements a flexible strategy which normally chooses the maximum sequence of flips followed by a final pivot (*i.e.*, the pivot is chosen to maximise the improvement in the dual objective). If it detects that the magnitude of the primal variables has grown to a point where numerical accuracy may be compromised, it will switch to choosing the pivot variable to minimise the maximum infeasibility over the primal variables.

The strategy used for generalised dual pivoting is controlled by the same option used to choose between standard and generalised dual pivoting. The complete set of options is standard dual pivoting; generalised dual pivoting to maximise dual objective improvement; generalised dual pivoting to minimise maximum primal infeasibility; and the flexible generalised strategy used as the default.

Antidegeneracy using perturbed subproblems is used with generalised dual pivoting. The alignment-based anti-degeneracy strategies are not implemented.

13 Primal Simplex

The primal simplex implementation in DYLP is a two-phase algorithm. DYLP will choose primal simplex phase II whenever the current basic solution is primal feasible but not dual feasible. It will choose primal simplex phase I when the current basic solution is neither primal or dual feasible. The primary role of primal simplex in DYLP is to reoptimise following the addition of variables. Since primal phase I requires neither primal or dual feasibility, it is the fallback simplex.

The primal simplex implementation incorporates projected steepest edge (PSE) pricing (§4.1), standard (§13.6) and generalised (§13.7) pivoting, and perturbation-based (§5) and alignment-based (§6) antidegeneracy algorithms.

Figure 9 shows the call structure of the primal simplex implementation.

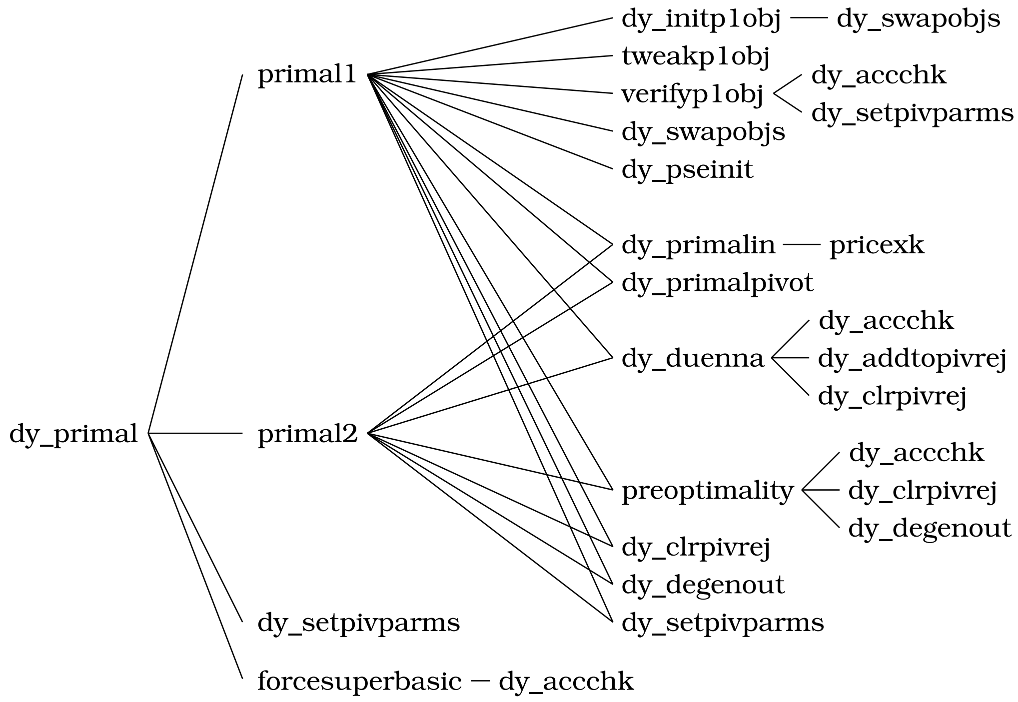

Figure 9: Call Graph for Primal Simplex

13.1 Primal Top Level

Primal simplex is executed when the dynamic simplex state machine enters one of the states dyPRIMAL1 or dyPRIMAL2. If required, the PSE reference frame is initialised to the nonbasic variables and the projected column norms are initialised to one (*vid.* §4.1), and the primal simplex routine dy_primal is called.

dy_primal controls the use of phase I (primal1) and phase II (primal2) of the primal simplex algorithm. The primary purpose of dy_primal is to provide a loop which allows a limited number (currently hardwired to 10) of reversions to phase I if primal feasibility is lost during phase II.

Loss of primal feasibility is treated as a numeric accuracy problem; with each such reversion the minimum pivot selection tolerances are tightened by one step.

To maintain primal feasibility when repairing a singular basis (§7.2) in primal phase II, superbasic variables may be created. Superbasic variables will not normally be created during phase I and the code assumes that it will not encounter them[4]. Rarely, a sequence of errors during phase II will cause DYLP to lose primal feasibility and revert to phase I with superbasic variables still present in the nonbasic partition. The routine forcesuperbasic is called to ensure that any superbasic variables are forced to bound in such a phase II to phase I transition.

13.2 Primal Phase I

The overall flow of phase I of the primal simplex is shown in Figure 10. The body of the routine is structured as two nested loops. The outer loop handles startup and termination, and the inner loop handles the majority of routine pivots. A pivot iteration in phase I normally consists of three steps: the actual pivot and variable updates, routine maintenance checks, and revision of the objective.

A dynamically modified artificial objective is used to guide pivoting to feasibility during phase I. The (minimisation) coefficients assigned to variables are -1 for variables below their bound, 0 for variables within bounds, and +1 for variables above their bound. On entry to phase I, dy_initp1obj forms a working set containing all infeasible variables, constructs the corresponding objective, swaps out the original objective, and installs the phase I objective.

Once the phase I objective has been constructed, the outer loop is entered and dy_primalin is called to select the initial entering variable. Then the inner loop is entered and dy_primalpivot is called to perform the pivot. dy_primalpivot (*vid.* §13.4) will choose a leaving variable (primalout), pivot the basis (dy_pivot), update the primal and dual variables (primalupdate), and update the PSE pricing information and reduced costs (pseupdate). For a routine pivot, pseupdate will also select an entering variable for the next pivot. dy_duenna evaluates the outcome of the pivot, handles error detection and recovery where possible, and performs the routine maintenance activities of accuracy checks and refactoring of the basis.

As the final step in a routine pivot, tweakp1obj scans the working set and removes any newly feasible variables. The objective function is adjusted to reflect any changes and reduced costs and dual variables are adjusted or recalculated as required. If there are no problems, the pivoting loop iterates, using the leaving variable selected in pseupdate. The loop continues until primal feasibility is reached, the problem is determined to be infeasible, or an exception or fatal error occurs.

When the working set becomes empty, tweakp1obj will give a preliminary indication of primal feasibility. If verifyp1obj confirms that all variables are primal feasible, the pivoting loop will end. If accumulated numerical inaccuracy has caused previously feasible variables to become infeasible, the pivot selection parameters will be tightened, dy_initp1obj will be called to build a new working set and objective, and pivoting will resume.

Changes to the objective coefficients may make it necessary to select a new entering variable. This situation arises when a variable gains feasibility but remains basic, as changing an entry of c^B can potentially affect all reduced costs[5]. The variable selected in pseupdate may no longer

[4] More strongly, superbasic variables are introduced only in primal phase II for the purpose of maintaining feasibility during repair of a singular basis. They will appear outside of primal2 only if the problem is unbounded or if primal2 terminates with an error condition.

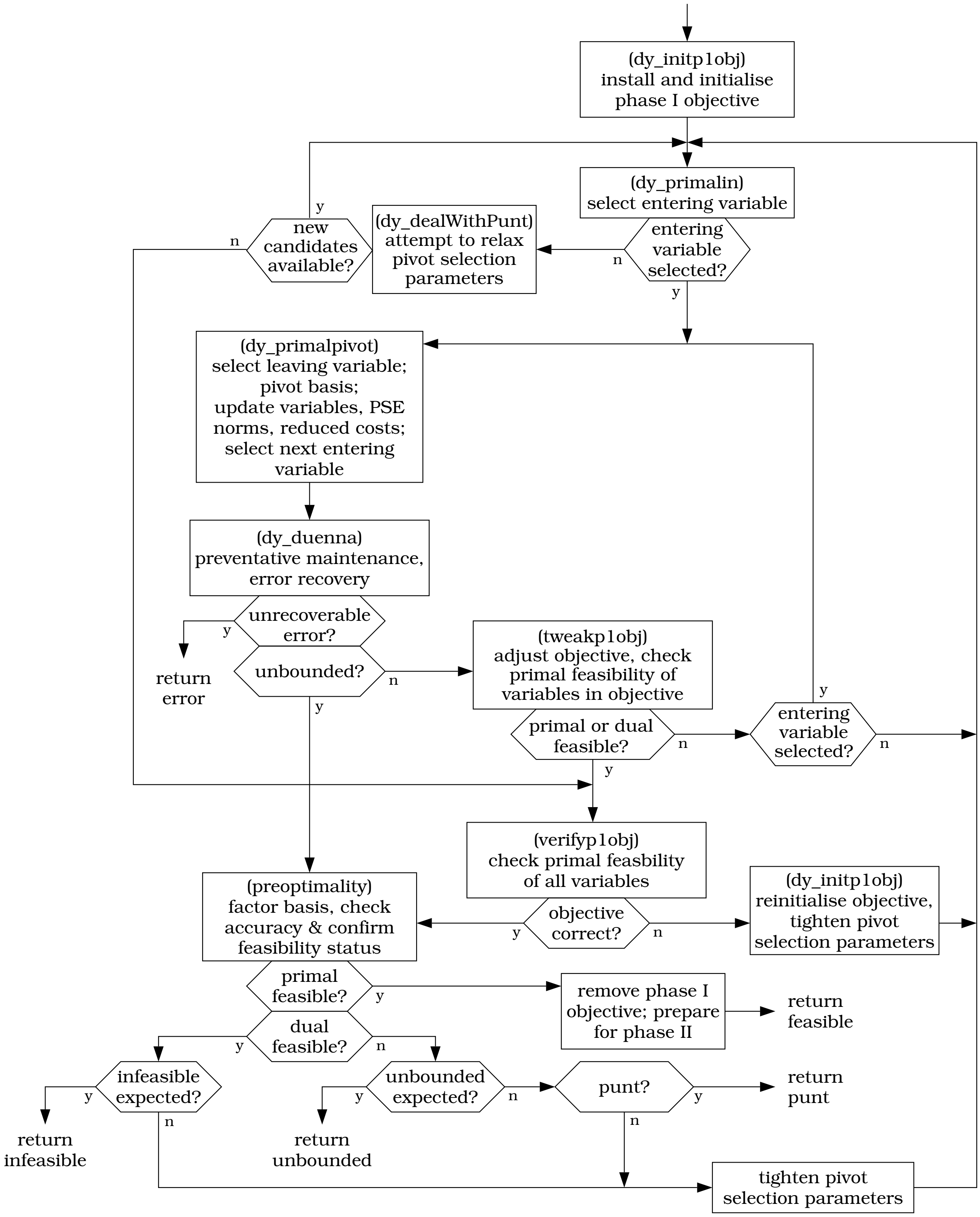

Figure 10: Primal Phase I Algorithm Flow

be the best (or even a good) choice. The flow of control is redirected to the outer loop, where dy_primalin will be called to select an entering variable.

It can happen that no entering variable is selected by pseupdate for use in the next iteration. Here, too, control flow is redirected to dy_primalin. The single most common reason in primal simplex is a bound-to-bound 'pivot' of a nonbasic variable — since there is no basis change, pseupdate is not called.

Another common reason for failure to select an entering variable is that all candidates were previously flagged as unsuitable pivots. In this case, dy_primalin will indicate a 'punt' and dy_dealWithPunt will be called to reevaluate the flagged variables. If it is able to make new candidates available, control returns to dy_primalin for another attempt to find an entering variable. If all flagged variables remain unsuitable, control flow moves to the preoptimality actions with an indication that primal phase I has punted.

If the current pivot is aborted due to numerical problems (an unsuitable pivot coefficient being the most common of these), pseupdate is not executed. Once dy_duenna has taken the necessary corrective action, the flow of control moves to the outer loop and dy_primalin.

When dy_primalin indicates optimality, dy_primalpivot indicates optimality or unboundedness, or tweakp1obj indicates primal feasibility, the inner pivoting loop ends and verifyp1obj is called to verify feasibility. If feasibility is confirmed, preoptimality is called to refactor the basis, perform accuracy checks, and confirm primal and dual feasibility. If there are no surprises, primal phase I terminates with an indication of optimality (primal feasibility), unboundedness, or primal infeasibility. In any event, if preoptimality reports that the solution is primal feasible, phase I will end with an indication of optimality even if it was not expected from the pivot loop termination condition.

If a primal feasible solution has been found, the original objective will be restored before returning from primal1. The transition to phase II entails calculating the objective, dual variables, and reduced costs for the original objective. If the problem is infeasible or unbounded, the phase I objective is left in place and DYLP will use it as it attempts to activate variables or constraints to deal with the problem (§11.2).

Loss of primal feasibility can occur when the basis is factored during the preoptimality checks. The pivot selection parameters are tightened and pivoting resumes.

Loss of dual feasibility is considered only when it is accompanied by lack of primal feasibility (*i.e.*, a false indication of infeasibility). Loss of dual feasibility can occur for two distinct reasons. In the less common case, loss of dual feasibility stems from loss of numeric accuracy. The pivot selection rules are tightened and pivoting resumes.

The more common reason for apparent loss of dual feasibility at the termination of phase I primal simplex is that it is ending with a punt, as described above. The variables flagged as unsuitable for pivoting are not dual feasible, and when the flags are removed to perform the preoptimality checks, dual feasibility is revealed as an illusion. No further action is possible within primal simplex; the reader is again referred to §11.2.

When the number of false indications of optimality exceeds a hard-coded limit (currently 15), primal simplex terminates with a fatal error. Other errors also result in termination of the primal simplex algorithm, and ultimately in an error return from DYLP.

[5] Less commonly, the problem arises because the newly feasible leaving variable of the just-completed pivot has been selected to reenter. The objective coefficient for this variable is incorrect when it is used by pseupdate.

13.3 Primal Phase II

The overall flow of phase II of the primal simplex is shown in Figure 11. The major differences

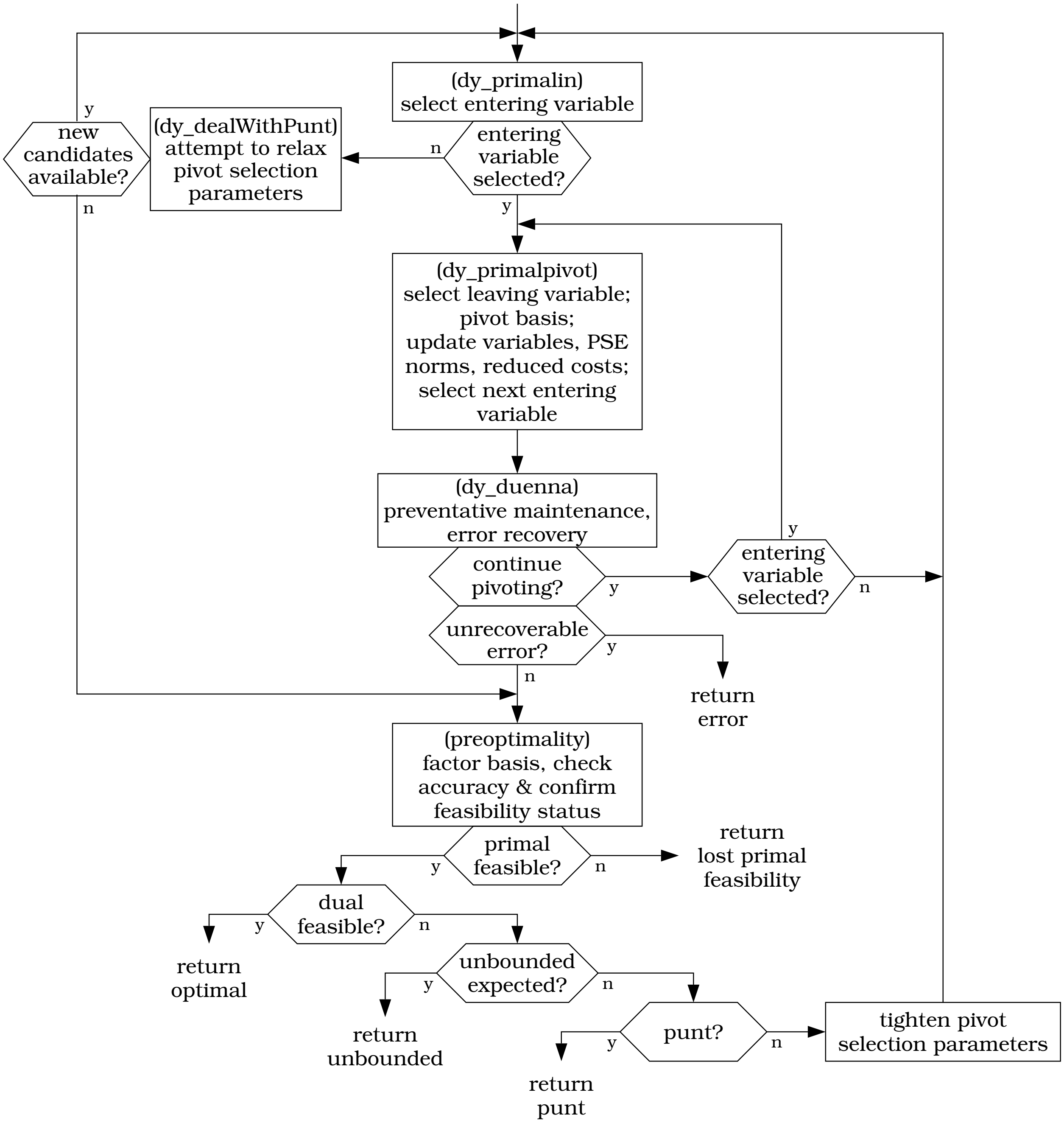

Figure 11: Primal Phase II Algorithm Flow

from phase I are that the problem is know to be feasible and the original objective function is used instead of an artificial objective function. This considerably simplifies the flow of primal2.

The inner pivoting loop has only two steps: the pivot itself (dy_primalpivot) and the maintenance and error recovery functions (dy_duenna). When dy_primalin indicates optimality or dy_primalpivot indicates optimality or unboundedness the inner loop ends and preoptimality is called for confirmation. preoptimality will refactor the basis, perform accuracy checks, recompute the primal and dual variables, and confirm primal and dual feasibility. If there are no surprises, primal phase II will end with an indication of optimality or unboundedness.

Loss of dual feasibility (including punts) is handled as described for primal phase I. Loss of primal feasibility causes primal2 to return with an indication that it has lost primal feasibility, and dy_primal will arrange a return to primal phase I.

13.4 Pivoting

DYLP offers two flavours of primal pivoting: A standard primal pivot algorithm in which a single primal variable is selected and pivoted into the basis, and an extended primal pivot algorithm which allows somewhat greater flexibility in the choice of leaving variable. By default, DYLP will use the extended algorithm.

Figure 12 shows the call structure of the primal pivot algorithm. The routine primalout implements standard primal pivoting; primmultiout implements extended primal pivoting.

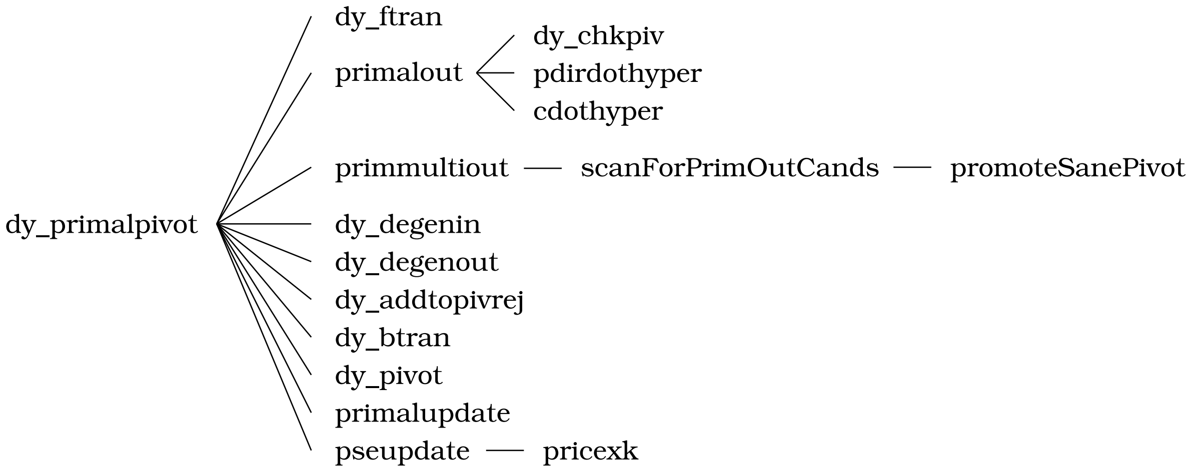

Figure 12: Call Graph for Primal Pivoting

The first activity in dy_primalpivot is the calculation of the coefficients of the pivot column, $\overline{a}_j = B^{-1}a_j$, by the routine dy_ftran. With the entering primal variable and the ftran'd column in hand, one of primalout or primmultiout are called to select the leaving variable.

If the entering and leaving variables are the same (*i.e.*, a nonbasic variable is moving from one bound to the other), all that is required is to call primalupdate to update the values of the primal variables. The basis, dual variables, reduced costs, and PSE pricing information are unchanged.

If the entering and leaving variables are distinct, the pivot is performed in several steps. Prior to the pivot, the i^{th} row of the basis inverse, β_i, and the vector $\tilde{a}_j^\top B^{-1}$ are calculated for use during the update of the PSE pricing information. The basis is pivoted next; this involves calls to dy_ftran and dy_pivot, as outlined in §7.3. If the basis change succeeds, the primal and dual variables are updated by primalupdate using the iterative update formulæ of §3, and then the PSE pricing information and reduced costs are updated by pseupdate, using the update formulæ of §4.1. As a side effect, pseupdate will select an entering variable for the next pivot.

13.5 Selection of the Entering Variable

Selection of the entering variable x_j for a primal pivot is made using the primal steepest edge criterion described in §4.1. As outlined above, the normal case is that the entering variable for the following pivot will be selected as pseupdate updates the PSE pricing information for the current pivot. In various exceptional circumstances where this does not occur, the routine dy_primalin is called to make the selection.

13.6 Standard Primal Pivot

Selection of the leaving variable x_i is made using standard primal pivoting rules and a set of tie-breaking strategies.

Abstractly, we need to check $x_k = \overline{b}_k - \overline{a}_{kj}\Delta_{kj}$ to find the maximum allowable Δ_{kj} such that $l_k \leq x_k \leq u_k \; \forall k \in B$ and $x_i = l_i$ or $x_i = u_i$ for some i. The index i of the leaving variable will be

$$i = \arg\min_k \left| \frac{\overline{b}_k}{a_{kj}} \right|$$

for suitable $x_k \in B$.

The primal pivoting rules are the standard set for revised simplex with bounded variables, and are summarised in Table 2. During phase I, when a variable is infeasible below its lower bound

leaving x_i	entering x_j	pivot $\overline{a}_{ij}$
$\nearrow ub$	$lb \nearrow$	< 0
	$ub \searrow$	> 0
$\searrow lb$	$lb \nearrow$	> 0
	$ub \searrow$	< 0

Table 2: Summary of Primal Simplex Pivoting Rules

and must increase to become feasible, DYLP sets the limiting Δ_j based on the upper bound, if it is finite, and uses the lower bound only when the upper bound is infinite. Similarly, when a variable must decrease to its upper bound, the lower bound is used to calculate the limiting Δ_j if it is finite.

DYLP provides a selection of tie-breaking strategies when there are multiple candidates with equal $|\Delta_{kj}| = \Delta_{\min}$. The simplest is to select the first variable x_k such that $\Delta_{kj} = 0$. A slightly more sophisticated strategy is to scan all variables eligible to leave and pick x_i such that $i =$

$\arg\max_{k \in K} |\overline{\alpha}_{kj}|$, $K = \{k \mid |\Delta_{kj}| = \Delta_{\min}\}$; DYLP will use this strategy by default. DYLP also provides four additional strategies based on hyperplane alignment, as described in §6. An option allows the tie-breaking strategy to be selected by the client.

In case of degeneracy, the perturbed subproblem anti-degeneracy algorithm described in §5 is also available. The client can control the use of perturbed subproblems through two options which specify whether a perturbed subproblem can be used, and how many consecutive degenerate pivots must occur before the perturbed subproblem is created. By default, DYLP uses perturbed subproblems aggressively and will introduce one when faced with a second consecutive degenerate pivot.

13.7 Extended Primal Pivot

All dual variables have a single finite bound of zero, so it's not possible to develop a generalised primal pivoting algorithm analogous to the dual pivoting algorithm of §12.6. It is, however, possible to introduce some flexibility in the selection of the leaving variable. We can also apply the same strategy used in generalised dual pivoting to promote a numerically stable pivot candidate over an unstable candidate.

In phase I, for an infeasible basic variable with finite upper and lower bounds, there are two points where the variable can be pivoted out of the basis: When the variable moves from infeasibility to one of its bounds (the 'near' bound), and when it has crossed the feasible region to the opposite ('far') bound. Pivoting when the near bound is reached is optional; pivoting at the far bound is mandatory if primal feasibility is to be maintained. The same notion can be applied in phase II, but its utility is much more limited: In cases where a basic variable is at its near bound and could be pushed to the far bound, we may prefer to choose a degenerate and numerically stable pivot over a degenerate and numerically unstable pivot.

DYLP implements extended primal pivoting by first collecting the set of candidates x_i to leave the basis. Variables with two finite bounds get two entries, one with the value of Δ_{ij} associated with the near bound, the other the value associated with the far bound. The set is then sorted using nondecreasing value of $|\Delta_{kj}|$, with numerical stability as the tie-breaker.

The process of scanning for candidates and sorting the resulting set is implemented in the routines scanForPrimalOutCands and primalcand_cmp. For efficiency, scanForPrimalOutCands keeps a 'best candidate' using the standard primal pivoting rules. If this candidate is good (nondegenerate and numerically stable), it is accepted as the leaving variable and no further processing is required.

If a good candidate is not identified by the scan, an attempt is made to promote a good candidate to the front of the sorted list. The criteria is as outlined for generalised dual pivoting: If the amount of primal infeasibility that would result from promoting a stable, nondegenerate candidate is tolerable, that candidate is promoted and made the leaving variable. This promotion of a stable pivot over an unstable pivot is implemented in the primal version of promoteSanePivot.

Antidegeneracy using perturbed subproblems is used with extended primal pivoting. The alignment-based anti-degeneracy strategies are not implemented.

14 Variable Management

Activation and deactivation of variables and constraints is a core activity for dynamic simplex. The activation or deactivation of variables can occur as an independent activity or as a consequence of constraint activation and deactivation (*vid.* §15). During normal execution (*vid.* Fig. 3) variables are activated (dy_activateVars) when primal simplex returns an indication of infeasibility or when primal or dual simplex achieve optimality. Variables are deactivated (dy_deactivateVars) when dual simplex achieves optimality and returns to primal phase II after adding variables.

In a somewhat different context, dual feasible variables are evaluated as dual bounding constraints and activated (dy_dualaddvars) when dual simplex indicates an unbounded dual (infeasible primal). Dual feasible variables are also activated (dy_activateVars) when dual simplex will be reentered after adding constraints without an intervening primal simplex phase. The motivation is to increase the probability that the dual problem will remain bounded.

Figure 13 shows the call structure for the top-level variable activation and deactivation routines.

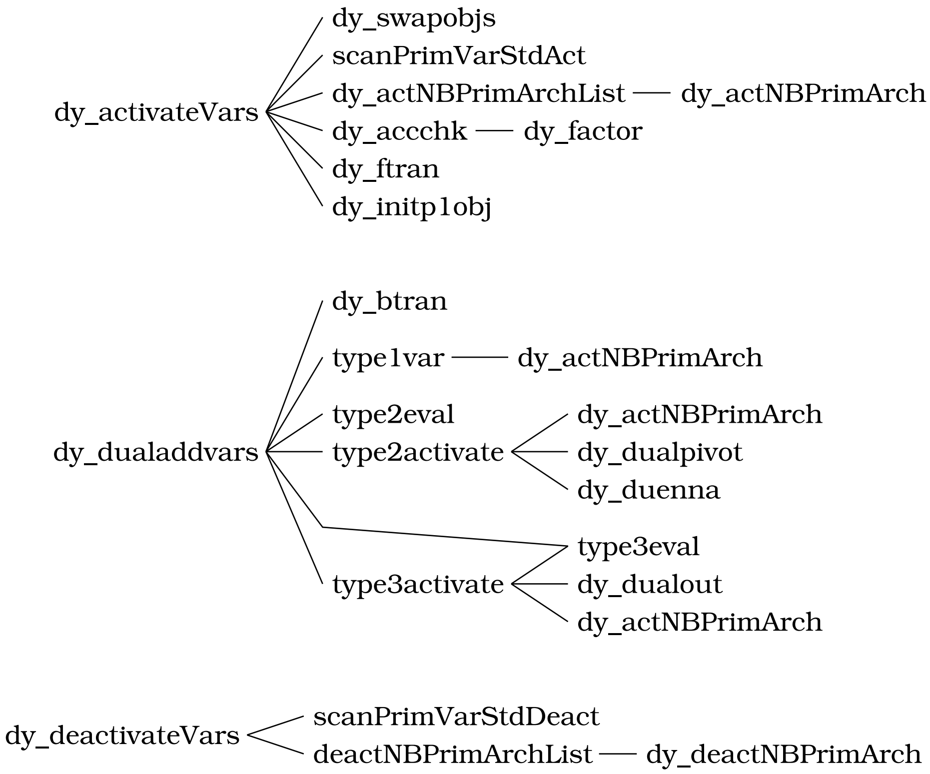

Figure 13: Call Graph for Variable Management Routines

14.1 Variable Management Primitives

There are two primitive variable management routines:

- dy_actNBPrimArch activates a primal architectural variable into the nonbasic partition.

- dy_deactNBPrimArch deactivates a nonbasic primal architectural variable.

DYLP assumes that inactive variables are feasible and at bound and provides no independent way to specify the value of the variable. As a special case, inactive free variables are assumed to have the value zero. A consequence of this is that it is not possible to deactivate a basic variable; the variable must first be forced into the nonbasic partition. Unless the variable is basic at bound, this will change the variable's value. The special-purpose routine dy_deactBPrimArch performs this service when DYLP is attempting to force primal feasibility by deactivating infeasible basic variables.

DYLP provides no method for activating an architectural variable into the basic partition. When activating a constraint, the logical variable associated with the constraint is always used as the new basic variable.

14.2 Activation of Variables

DYLP looks for variables to activate whenever optimality is attained for the current set of constraints and variables, or when the active system is found to be infeasible. The set of inactive variables is scanned and any variables with favourable reduced costs are activated and placed in the primal nonbasic partition.

If an optimal solution has been found for the active constraint system by either primal or dual simplex, scanPrimVarStdAct is called to select a set of variables to be activated under the assumption that primal phase II iterations will resume after the variables are added. The reduced costs are calculated using the original objective function for the problem. Variables are selected for activation if their reduced cost indicates they are not at their optimal bound (*i.e.*, dual infeasible).

If phase I of the primal simplex has found the problem to be infeasible, scanPrimVarStdAct is again used to select the set of variables to be activated, but the reduced costs are calculated using the phase I objective (as described in §13.2). Primal phase I iterations resume after variables are added.

Normally, when dual simplex indicates optimality, primal phase II is executed after adding variables with favourable (dual infeasible) reduced costs. It can happen, however, that there are no such variables. In this case, DYLP will attempt to add violated constraints and, if any are found, resume execution of dual simplex. To increase the likelihood that the dual problem will remain bounded, DYLP will again attempt to add variables before resuming dual simplex iterations, but the criteria in this case will be variables whose reduced costs are dual feasible (*i.e.*, unfavourable from a primal perspective).

Activating a variable into the nonbasic partition will not change to the basis, primal or dual variable values, or DSE pricing information. The reduced cost and the projected column norm used for PSE pricing must be properly initialised for the new variable. The action taken for the projected column norm depends on the context of variable activation. If primal simplex was executing prior to variable activation and will be resumed after variable activation, the projected column norms are up-to-date and correct values must be calculated for the new variables. In other cases, PSE pricing information will be initialised when primal simplex iterations resume and no action is required.

If the dual simplex has found the problem to be primal infeasible (dual unbounded), the problem of selecting variables to add should be viewed from the perspective of looking for dual constraints which will bound the problem. The goal is to activate one or more dual constraints and return to dual simplex iterations.

The selection of the candidate entering dual variable y_i (leaving primal variable x_i) has fixed the direction of travel, ζ_i. The best outcome will be to add dual constraints (primal variables) which block travel in the direction ζ_i. If that isn't possible (because activating any bounding dual constraint would result in the loss of dual feasibility) a second possibility is to activate variables which will change the dual reduced costs (the values of the primal basic variables) so that a different dual variable y_k is selected to enter. The hope is that motion in a different direction ζ_k may make it possible to activate constraints which will bound the dual without loss of feasibility.

The subroutine dy_dualaddvars controls the search process, and can activate three classes of variables, for convenience called type 1, type 2, and type 3.

Type 1 variables are those variables which constitute feasible dual constraints which bound the dual problem. These can be activated and placed in the primal nonbasic partition without losing dual feasibility. Type 1 variables are preferred, as dy_dualaddvars can activate any number of them in a given call.

If there are no type 1 variables, dy_dualaddvars considers type 2 variables. Type 2 variables are those variables which constitute dual constraints that bound the dual problem and which, while not dual feasible if activated into the primal nonbasic partition, will give a dual feasible solution if activated and immediately pivoted into the basis. This is equivalent to adding a cutting plane which renders the current solution infeasible and executing a single pivot to regain feasibility; necessarily, the objective will deteriorate. In the context of Table 1 in §12.5, this amounts to selecting a pivot with the signs of $\overline{c}_j$ and $\overline{a}_{ij}$ reversed. The pivot is sufficiently similar to a normal dual pivot that it can be handled by dy_dualpivot. It is not standard in that the entering primal variable will move away from its bound toward the infeasible side (*e.g.*, x_j would enter falling from its lower bound with $\overline{c}_j < 0$ and $\overline{a}_{ij} > 0$). One such variable can be activated on each call to dy_dualaddvars.

In the absence of type 1 or type 2 variables, type 3 variables are considered. These are variables which are not dual feasible at their current bound but which will reduce the infeasibility of the leaving primal variable if activated and changed to their opposite bound. The motivation for activating a type 3 variable is that it makes the reduced cost of y_i less desirable, so that some other variable y_k can be selected to enter (thus moving in a different direction ζ_k). The routine type3activate will attempt to activate as many type 3 variables as required in order to change the entering dual variable y_i.

Activation of type 2 or type 3 variables is generally not cost-effective. By default, DYLP limits dy_dualaddvars to type 1 activations. The dynamic simplex algorithm will revert to primal phase I if no type 1 variables exist. An option allows the client to specify whether type 1, type 2, or type 3 variables will be considered.

Activation of a type 1 variable is no different from any other activation into the nonbasic partition, as described above. For type 2 variables, the pivot will cause a change of basis. dy_dualpivot will take care of the required calculations and updates in the context of dual simplex. For type 3 variables, the basis doesn't change, and the values of the dual variables and DSE norms are unchanged. The values of the primal variables do change, however, and this changes the DSE pricing information.

14.3 Deactivation of Variables

Deactivation of variables occurs when dual simplex finds an optimal solution for the active constraint system and variable activation identifies dual infeasible variables for activation. In this case, variable deactivation is performed before entering primal phase II simplex. The subroutine dy_deactivateVars is called to deactivate variables according to a client-specified threshold, expressed as a percentage of the maximum unfavourable reduced cost over all active variables.

Specifically, dy_deactivateVars scans the reduced costs of the active variables and determines a pair of values $\check{c} = \max_{\{k:c_k<0\}} |c_k|$ and $\hat{c} = \max_{\{k:c_k>0\}} |c_k|$. It then deactivates variables with $c_k > \hat{c}$(dy_tols.purgevar) or $c_k < -\check{c}$(dy_tols.purgevar).

14.4 Initial Variable Selection

For a cold start, the initial set of active variables is completely determined by the initial set of constraints. All variables referenced in the constraints are activated. As noted in §10, the client can set parameters which will cause variable deactivation to be executed prior to starting simplex iterations.

For a hot start, the initial set of active variables is the set that was active at return from the previous call to dylp.

For a warm start, the set of active constraints is specified by the basis. The initial set of active variables can be determined from the constraints as for a cold start, or the client can specify a set of variables which should be activated as the active constraint system is created.

As noted in §10, for a hot or warm start the client can set parameters which will cause variable activation to be executed prior to starting simplex iterations.

15 Constraint Management

Constraint management activities can be separated into selection of the initial constraint set, activation of violated or bounding constraints, and deactivation of loose constraints. In general, the goal is to maintain an active constraint system which is a subset of the original constraint system, consisting only of equalities and those inequalities necessary to define an optimal extreme point. DYLP expects that all constraints will be equalities or ≤ inequalities. Figure 14 shows the call structure for the constraint activation and deactivation routines.

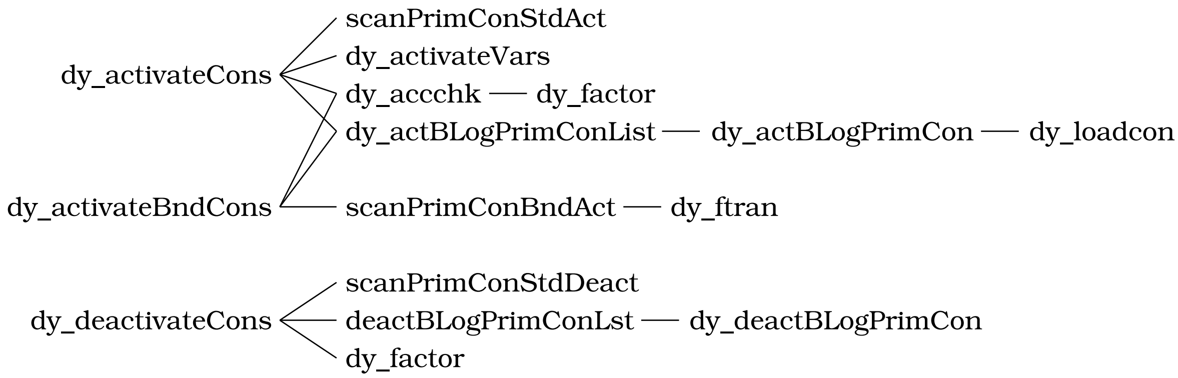

Figure 14: Call Graph for Constraint Management Routines

During construction of the initial constraint system, any variables referenced in a constraint are activated along with the constraint. During subsequent constraint activation phases, variable activation is more selective. The logical variable for the constraint is created and used as the new basic variable. If the next simplex will be primal simplex, activation is restricted to the subset of referenced variables with dual infeasible (favourable) reduced cost. If the next simplex will be dual simplex, activation is restricted to the subset of referenced variables that are dual feasible.

When a constraint is deactivated, only the slack variable for the constraint is deactivated. This minimises the work that must be performed to repair the basis.

15.1 Initial Constraint Selection

For a warm or hot start, the initial active constraint system is completely determined from the basis supplied by the client. As noted in §10, the client can set parameters which will cause constraint activation to be executed prior to starting simplex iterations. In this specific case, variable activation is not automatic and must be requested independently if desired.

For a cold start, where no initial basis is supplied, the initial active constraint system will include all equalities and a client-specified selection of inequalities. See §10.1 for a more detailed description.

15.2 Activation of Constraints

DYLP enters the constraint activation phase whenever the system is found to be primal unbounded or optimal for the set of active constraints and all variables (active and inactive). When the system is found to be optimal, DYLP calls dy_activateCons to search the inactive constraints for violated constraints. When the system is found to be unbounded, DYLP first calls dy_activateBndCons to search the inactive constraints for feasible constraints which block the direction of recession. If such bounding constraints exist, they are activated and primal phase II simplex is resumed. Otherwise, dy_activateCons is called to add any violated constraints and execution will go to primal phase I or dual simplex as appropriate.

Violated constraints are identified using a straightforward scan of the inactive constraints. The routine scanPrimConStdAct evaluates each constraint at the current value of x and returns a list of violated constraints. The routines dy_activateBLogPrimConList and dy_activateBLogPrimCon perform the activations. Following activations, the logical variables for the new constraints are made basic, the basis is refactored, and a new basic solution is calculated. If the call to dy_activateCons requested activation of referenced variables, dy_activateBLogPrimConList will collect a set of variable indices for activation. After the basis has been refactored, the set is passed to dy_activateVars for activation. If dual simplex will be the next simplex executed, only dual-feasible variables are activated.

In DYLP, unboundedness is detected by the primal simplex implementation; dual simplex is not called until primal simplex has found an initial optimal solution. When unboundedness is discovered, DYLP calls dy_activateBndCons to search for bounding constraints which are feasible at the current basic solution. A constraint will block motion in the direction η_j if $a_i \cdot \eta_j > 0$ for x_j increasing, or $a_i \cdot \eta_j < 0$ for x_j decreasing. This scan is performed by scanPrimConBndAct. Once the list of constraints is returned, constraint activation and basis repair proceed as in the case of violated constraints, but referenced variables are not activated.

When a constraint is activated, the set of basic variables is augmented with the slack variable for the constraint. Because the slack is basic, the value of the associated dual is zero. The basis will change, but the values of other active primal variables will remain the same. Since the new slack variables are not part of the PSE reference frame, the projected column norms associated with PSE pricing are unchanged. Because the objective coefficients associated with the slack variables are 0, the values of the preexisting dual variables and the reduced costs remain unchanged.

15.3 Deactivation of Constraints

Constraint deactivation is handled by dy_deactivateCons. DYLP implements three options for constraint deactivation, 'normal', 'aggressive', and 'fanatical'. When normal constraint deactivation is specified, DYLP will only deactivate inequalities which are strictly loose. Eligible inequalities are identified by scanning the basis for slack variables which are strictly within bounds. When aggressive constraint deactivation is specified, DYLP will also deactivate tight inequalities whose associated dual variable is zero. When fanatical constraint deactivation is specified, DYLP will deactivate any constraint (equality or inequality) whose associated dual is zero. The set of constraints to be deactivated is identified by the routine scanPrimConStdDeact.

Once a set of constraints has been identified for deactivation, the routines deactBLogPrimConList and dy_deactBLogPrimCon are called to perform the deactivations. The corresponding constraint is deactivated and removed from the active constraint system along with its associated logical variable. The basis is patched, if necessary, by moving the variable which is basic in the position of the deactivated constraint to the basis position which was occupied by the constraint's associated logical.

As with activation of constraints, deactivation of constraints changes the basis and DYLP will refactor and recalculate the primal and dual variables. The dual variables do not change, nor do the reduced costs of the remaining variables, since the cost coefficient of a logical variable is zero. In general, the PSE column norms will be changed because the deleted logical variables may be part of the reference frame. DYLP opts to reset the reference frame to deal with this, rather than updating or recalculating the column norms.

16 DYLP Interface

This section describes the native interface for DYLP. In addition to the main routine, dylp, various pricing, printing, and utility routines are provided. These routines, and the major interface structures, are described briefly in this section. For additional details on how to use DYLP, consult the comments in the source, particularly in dylp.h and dy_setup.c, and the example drivers supplied in the distribution.

DYLP's native interface is peculiar to DYLP and a bit low-level in places. Many individuals will find it more convenient to use DYLP as an embedded component within the software infrastructure provided by the COIN-OR project [2]. For details of the COIN OSI layer for DYLP, OsiDylp, please consult the COIN documentation. An added advantage of this approach is that the COIN OSI API provides a solver-independent interface. The underlying solver can be easily changed because the OSI layer insulates the client from the details of the solver's native interface.

The DYLP distribution provides a simple C driver program using DYLP's native interface in the file osi_dylp.c. The command 'osi_dylp -h' will print a message describing the available command line options.

DYLP assumes that the constraint system passed to it as a parameter *does not* contain logical variables (*i.e.*, slacks and artificials). On occasion, it must return values for logical variables; in such cases, it will use the negative of the index of the associated constraint.

16.1 Simplex Solver

DYLP is called as

lpret_enum dylp (lpprob_struct *orig_lp, lpopts_struct *orig_opts, lptols_struct *orig_tols, lpstats_struct *orig_stats)

> The orig_lp structure (§16.9) specifies the constraint system, control options, and (optionally) an initial basis and status vector and an initial active variable set. It is used to return the final status, primal and dual variable values, basis, and status vector, and (optionally) the active variables.
>
> The orig_opts structure (§16.10) specifies option settings to control DYLP's actions. The orig_tols structure (§16.11) specifies numeric tolerances and related control information.
>
> The optional structure orig_stats (§17) can be used (in conjunction with conditionally compiled code) to return detailed statistics about DYLP's actions.

16.2 Parameter Routines

The normal sequence to establish parameter values for DYLP is as follows:

1. The client calls dy_defaults to allocate option and tolerance structures and populate them with default values. The client can then adjust the parameters as desired.
2. The client somehow establishes the original copy of the constraint system. Typically, this will be a call to a constraint system generator[6], or a call to a routine which will read an MPS file.

3. The client calls dy_checkdefaults to to set parameter values which are calculated based on properties of the constraint system, and to ensure that all parameters are within acceptable bounds.

void dy_defaults (lpopts_struct **opts, lptols_struct **tols)

This routine will allocate an options structure opts and a tolerance structure tols and populate them with the standard default values for DYLP. Note that default values for some parameters are calculated in dy_checkdefaults based on the size of the constraint system.

void dy_checkdefaults (consys_struct *sys, lpopts_struct *opts, lptols_struct *tols)

This routine checks limits on parameter values and calculates values which depend on the size of the constraint system. User-supplied values are *not* overridden unless they are outside of DYLP's bounds for the parameter.

void dy_setprintopts (int lvl, lpopts_struct *opts)

This routine is provided purely for convenience; it will set all of DYLP's print levels based on the single value supplied for lvl. Roughly, lvl = 0 suppresses all output, lvl = 1 establishes the default print levels, which allow messages about extraordinary events, and lvl ≥ 2 provides increasing amounts of information. Consult the code for details.

16.3 Basis Package Initialisation

The GLPK basis package used in DYLP maintains static data structures that must be initialised before use and freed after use. For efficiency, it is useful to postpone initialisation until the size of the constraint system is known and can be used to estimate the size of the basis package's data structures, but DYLP will expand the basis structures if it detects that the constraint system has grown too large for the allocated capacity. Initialisation must occur before the first call to dylp. The basis structures should be freed when they are no longer needed.

void dy_initbasis (int concnt, int factor_freq, double zero_tol)

dy_initbasis initialises the data structures used by the GLPK basis maintenance package. concnt specifies the maximum allowable number of rows (constraints). factor_freq is the maximum number of basis updates which can occur between each (re)factorisation of the basis. A conservative value will be a bit larger than the regular refactorisation interval; for DYLP, lpopts.factor + 5. The final parameter, zero_tol, can be used to override GLPK's default zero tolerance if it is set to any value other than zero. Be sure you understand the ramifications of overriding the default.

The routine sets several other parameters important to pivoting. Interested readers should consult the comments in the code (dy_basis.c:dy_initbasis).

void dy_freebasis (void)

This routine will free the data structures allocated by the call to dy_initbasis.

[6]Consult the consys documentation for information on how to use the routines in the consys package to build a constraint system from scratch.

16.4 Information and Error Messages

DYLP uses private library packages for information and error messages[7]. The most visible value-added service provided by the libraries is integration of file and terminal output. Routines which generate output accept parameters to specify whether the output generated by a call should be sent to a file, to the terminal, both, or neither. The library packages must be initialised during startup. A brief explanation is provided here.

Information Messages

The I/O library provides a convenient means to generate information messages. Information messages may use any of the standard C conversion specifications; the underlying print engine for the current implementation is vfprintf. In addition to integrated file and terminal i/o, the library manages open file descriptors and coordinates activity with the error message library. See the code for examples of usage of the routines used to generate information messages (outchr, outfmt, and outfxd). The simple driver in osi_dylp.c contains a fragment of code which uses the chgerrlog routine to merge information and error messages in a single log file.

Initialisation and shutdown of the error message package is accomplished with the routines ioinit and ioterm.

bool ioinit (void)

Initialises internal data structures.

void ioterm (void)

Cleans up and shuts down the i/o package. Note that ioterm *does not* close open streams. It is assumed that the client will close open streams as appropriate, and that remaining streams can be left open until closed by the operating system at program termination.

Error Messages

The error message library provides a convenient means to generate warning and error messages. Error messages may use any of the standard C conversion specifications; the underlying print engine for the current implementation is vfprintf. The text of error messages reside in a file (bonsaierrs.txt in the DYLP distribution). Error messages are printed using the routines warn and errmsg. In calls to these routines, the error message is specified by a number. If an error message file cannot be located, a generic error message giving the error number will be produced. See the code for examples of usage of the routines used to generate warning (warn) and error (errmsg) messages.

Initialisation and shutdown of the error message package is accomplished with the routines errinit and errterm.

void errinit (const char *emsgpath, const char *elogpath, bool errecho))

The parameter emsgpath specifies the file containing the error messages. The parameter elogpath specifies a file name to be used to log error messages; if null, error messages are

[7]This usage is historical, rooted in an ancient era when i/o was still a roll-your-own enterprise that differed dramatically from one operating system and programming environment to the next.

not logged. The parameter errecho should be set to true if error messages should be echoed to stderr, false otherwise.

void errterm (void)

Cleans up and shuts down the error message package. In keeping with the behaviour of ioterm, it is left to the client or operating system to close any error log file.

On startup, the error message package should be initialised first, followed by the i/o package. At termination, the i/o package should be shut down first.

16.5 Summary of DYLP Startup and Shutdown

Pulling together the information from the previous sections, the sequence of actions required to use DYLP is listed below.

1. Initialise the error message and i/o packages. Open log files for information and error messages (optional).
2. Establish default parameter structures. Open and parse a file of DYLP option specifications (optional).
3. Create a constraint system using a constraint generator or by reading an input file. Adjust options and tolerances to match the constraint system. At some point between creating the constraint system and calling dylp, convert any '≥' constraints to other forms.
4. Initialise the basis package.
5. Construct parameter structures and call dylp.
6. Process the answer, restoring '≥' constraints and adjusting the answer appropriately, if the application demands it.
7. Free data structures. This may require an additional call to dylp, if the parameters given in the previous call instructed DYLP to retain internal data structures for efficient reoptimisation. It will certainly require calls to dy_freebasis, dy_freesoln, and consys_free.
8. Close files and shut down the i/o and error message packages.

Consult the sample drivers provided with DYLP for example implementations.

16.6 Pricing Routines

DYLP provides two additional routines which are useful in a mixed-integer linear programming environment. dy_pricenbvars will calculate the reduced cost for nonbasic variables and dy_pricedualpiv will calculate the cost of a dual pivot (a generalised penalty calculation).

bool dy_pricenbvars (lpprob_struct *orig_lp, flags priceme,
double **p_ocbar, int *p_nbcnt, int **p_nbvars)

This routine calculates the reduced cost of nonbasic variables, tapping the DYLP data structures for active variables and calculating the reduced cost as needed for inactive variables.

priceme provides limited additional control by allowing the client to specify the status of the nonbasic variables that should be priced. For example, to price all variables that are nonbasic at their upper or lower bound, priceme should be set to vstatNBUB | vstatNBLB. Other nonbasic variables (fixed, free, or superbasic) will not be priced. (See the section on status codes in dylp.h:for additional information.) The routine returns a compact list of p_nbcnt indices of priced variables in p_nbvars, with the corresponding reduced costs in p_ocbar. The indices returned in p_nbvars are the indices used in the original constraint system, which does not contain logical variables. Where nonbasic logical variables are present in the active system, they are identified in p_nbvars by the negative of the index of the associated constraint. In particular, the values returned are appropriate for use as the nbcnt, nbvars, and cbar parameters to dy_pricedualpiv.

bool dy_pricedualpiv (lpprob_struct *orig_lp, int oxindx, double nubi, double xi, double nlbi, int nbcnt, int *nbvars, double *cbar, double *p_upeni, double *p_dpeni)

This routine calculates the cost of the first dual pivot associated with forcing the value of the basic variable x_i down to a new upper bound u_i (a down penalty) or up to a new lower bound l_i (an up penalty).

The up penalty is $upen_i = \min_k \left(-(l_i - x_i)\dfrac{\overline{c}_k}{\overline{a}_{ik}}\right)$ for $\{k \in N \mid \overline{a}_{ik} < 0 \wedge x_k < u_k \vee \overline{a}_{ik} > 0 \wedge x_k > l_k\}$.

The down penalty is $dpen_i = \min_k \left(-(u_i - x_i)\dfrac{\overline{c}_k}{\overline{a}_{ik}}\right)$ for $\{k \in N \mid \overline{a}_{ik} > 0 \wedge x_k < u_k \vee \overline{a}_{ik} < 0 \wedge x_k > l_k\}$.

To perform the standard penalty calculation for forcing a basic variable to an integral value, the new lower bound would be $\lceil x_i \rceil$ and the new upper bound would be $\lfloor x_i \rfloor$. The basic variable x_i can be an architectural or a logical variable. The routine is capable of pricing a pivot involving the logical variable for a constraint that is not currently active.

oxindx specifies the basic variable to be priced (a logical is specified as the negative of the index of the associated constraint). xi is the current value of x_i in the optimal solution to the LP. (In the case of the logical for an inactive constraint, the value is obtained by evaluating the constraint at the current solution.) nubi is the new upper bound u_i, and nlbi is the new lower bound l_i. It should be true that nubi ≤ xi ≤ nlbi. nbcnt, nbvars, and cbar are as described for dy_pricenbvars. The up and down penalties will be returned in p_upeni and p_dpeni, respectively.

16.7 Print Routines

There are three routines to supply strings for DYLP status, phase, and return codes, a routine to print the compact solution returned by dylp, and a routine to print the contents of the statistics structure.

dy_dumpcompact (ioid chn, bool echo, lpprob_struct *soln, bool nbzeros)

This routine prints the solution returned by dylp in soln using a human-readable format. Output is directed to the channel specified by chn, and echoed to the terminal if echo is true. Normally, nothing is printed for nonbasic variables with a value of zero; set nbzeros to true to force them to be printed.

dy_dumpstats (ioid chn, bool echo, lpstats_struct *lpstats, consys_struct *orig_sys)

This routine prints the contents of the lpstats structure in a human-readable format. chn and echo are as for dy_dumpcompact. orig_sys should be the same constraint system referenced in the orig_lp parameter to dylp

dy_prtlpret (lpret_enum lpret)

Returns a pointer to a string for the return code specified in lpret.

dy_prtlpphase (dyphase_enum phase, bool abbrv)

Returns a pointer to a string for the return code specified in phase. If abbrv is true, this will be a two-character abbreviation.

dy_prtvstat (flags status)

Returns a pointer to a static buffer containing a string representation of the status flags specified in status. The buffer is overwritten at each call.

16.8 Utility Routines

An eclectic trio of additional interface routines.

bool dy_dupbasis (int dst_basissze, basis_struct **p_dst_basis, basis_struct *src_basis,
int dst_statussze, flags **p_dst_status, int src_statuslen, flags *src_status)

This routine will duplicate the basis and status arrays. Data structures will be allocated as required if they are not supplied as parameters.

bool dy_expandxopt (lpprob_struct *lp, double **p_xopt)

This routine will expand the compact form of the solution in lp into a single vector p_xopt. The vector will be allocated if one is not supplied as a parameter.

dy_freesoln (lpprob_struct *lpprob)

This routine will free the data structures used to hold the LP solution, including data structures for the basis, status vector, primal and dual variable values, and the active variables vector.

16.9 The LP Problem Specification

The structure lpprob_struct *orig_lp is used to define the LP problem to DYLP and to return the answer to the client. It holds pointers to the constraint system, an active variable vector, a basis vector, a status vector, and vectors for the primal and dual variables, as well as fields for information and control. Each field is discussed below; for precise details, the reader should consult the file dylp.h.

actvars A vector used to specify and/or return the set of active variables. The vector supplied as an input parameter will be overwritten on output.

(i) For a warm start, an initial set of active variables can be specified. This information will be used only if the lpctlACTVARSIN flag is set in the ctlopts field. For a cold or hot start, a vector can be provided to return the final set of active variables.

(o) The final set of active variables. If no vector was supplied as an input parameter, DYLP will allocate one on output. Active variable information is returned only if the lpctlACTVARSOUT flag is set when DYLP is called. Valid information is returned only if an optimal solution is found. If valid information is not returned, the lpctlACTVARSOUT flag will be reset.

basis A data structure for the LP basis. Because the set of active constraints at optimum will not, in general, include all constraints, the basis vector specifies the constraint and the primal variable in each basis position.

(i) For a warm start, an initial basis must be provided. For a cold or hot start, a structure can be provided to return the final basis.

(o) The final basis. If no vector was supplied as an input parameter, DYLP will allocate one on output.

colsze The allocated column capacity of the data structure. The status and actvars data structures, if provided by the client, must be capable of holding this many entries. If colsze is insufficient to return the answer, DYLP will reallocate the data structures.

consys The constraint system, in the format described for the CONSYS constraint system subroutine library [5].

ctlopts A vector of flags used to specify optional actions and status. The current set of flags can be used to control allocation and deallocation of internal DYLP data structures (lpctlDYVALID, lpctlNOFREE, lpctlONLYFREE), specify the presence of changes to the problem bounds (lpctlUBNDCHG, lpctlLBNDCHG, lpctlRHSCHG) and objective (lpctlOBJCHG), specify initial variable and/or constraint activation (lpctlINITACTVAR, lpctlINITACTCON), and specify the exchange of active variable information (lpctlACTVARSIN, lpctlACTVARSOUT).

iters The total number of simplex iterations.

lpret The return code from the simplex routine.

If no errors occur, the code should be one of lpOPTIMAL (optimal), lpINFEAS (primal infeasible), or lpUNBOUNDED (primal unbounded).

Error returns include lpPUNT (nonbasic variables exist with favourable reduced costs, but they cannot be pivoted due to unsuitable pivot coefficients), lpLOSTFEAS (primal feasibility has been lost and DYLP has exceeded its limit on attempts to regain feasibility), lpSTALLED (the limit on pivots without improvement in the objective has been exceeded, due to cycling or stalling), lpITERLIM (a limit on pivots per phase or total pivots has been exceeded), lpACCCHK (a numerical accuracy check has occurred), lpNOSPACE (the GLPK basis routines could not acquire sufficient space to maintain the basis inverse), lpFATAL (an unspecified fatal error has occurred), and lpINV (DYLP aborted due to internal confusion).

obj For an optimal result, the value of the objective function. For an infeasible result, the total primal infeasibility. For an unbounded result, the index of the unbounded variable, negated if the variable can decrease without bound, positive if it can increase without bound. For any other return status, this field is undefined.

phase

- **(i)** If the phase is set to dyDONE, DYLP will assume that the only purpose of the call is to free internal data structures. Other values are ignored.
- **(o)** The termination phase of the dynamic simplex algorithm; should be dyDONE unless an error has occurred, in which case it'll be dyINV.

rowsze The allocated row capacity of the data structure. The basis, x, and y data structures, if provided by the client, must be capable of holding this many entries. If rowsze is insufficient to return the answer, DYLP will reallocate the data structures.

status A data structure to hold the status of variables. For nonbasic variables, an entry is a DYLP status code (vstatNBFX, vstatNBUB, vstatNBLB, or vstatNBFR). For basic variables, an entry is the negative of the basis position.

- **(i)** For a warm start, an initial status must be provided. For a cold or hot start, a structure can be provided to return the final status.
- **(o)** The final status vector. The value of nonbasic primal variables is returned through this vector. If no vector was supplied as an input parameter, DYLP will allocate one on output.

x A data structure to hold the values of the basic primal variables.

- **(i)** A structure can be provided to return the final values.
- **(o)** The values of the basic primal variables, indexed by basis position. If no vector was supplied as an input parameter, DYLP will allocate one on output.

y A data structure to hold the values of the dual variables.

- **(i)** A structure can be provided to return the final values.
- **(o)** The values of the dual variables, indexed by basis position. If no vector was supplied as an input parameter, DYLP will allocate one on output.

16.10 DYLP Options

DYLP is intended to be a flexible testbed, and as such has a large number of options. Many, in fact, have argued that it has entirely too many options. The author offers two observations in his own defense:

- All of them, at some point, were useful to him, and
- if you're not interested, ignore them all and let DYLP choose what it thinks are reasonable values.

If you look through the code, you may notice a few options that aren't documented here. By and large, this is because the best choice is clear and choices other than the current default give uniformly poor performance.

Options are held internally in a lpopts_struct structure. Each field is described briefly below, including default values. The reader is encouraged to consult dylp.h for details, and dy_setup.c to confirm that default values have not changed since this documentation was written.

Most options can be set using commands read from an options file. This file is parsed by a simple command interpreter (contained in cmdint.c) and support routines in dy_setup.c and in the i/o library (*vid.* §16.4). If your application has some other way to acquire options from the user, all that's really necessary is a way to create and load a lpopts_struct to pass as a parameter to dylp. As described in §16.2, the routines dy_defaults and dy_checkdefaults will, respectively, initialise a lpopts_struct with default values and adjust those values to match the constraint system.

In the individual option descriptions which follow, the first line provides the syntax expected by the simple command interpreter mentioned above. information about acceptable values. Where applicable, for simple numeric parameters, the next line gives the lower bound, default value, and upper bound for the option in the notation (lower bound) ≤ (default value) ≤ (upper bound). The remainder of the entry describes the action of the option.

active: cons, vars

`lpcontrol active` *size-spec*-LIST′ `;`
size-spec ::= `variables` float | `constraints` float

$0.0 \leq .25 \leq 1.0$ for both

The values active.vars and active.cons specify the fraction of variables and constraints, respectively, which are expected to be active at any one time. The initial allocated capacity of the active constraint system data structure will be the specified fraction of the number of variables and constraints in the constraint system passed to dylp. They do not represent limits — the constraint system will be expanded as required. They are exposed for efficiency in the event that the client can provide a better estimate for the expected size of the active constraint system.

Note that specifying active.vars = 1.0 and active.cons = 1.0 is *not* the same as specifying that DYLP use the full constraint system (*cf.* fullsys). The data structure for the active constraint system will be created with the capacity to hold the full constraint system, but constraint and variable activation and deactivation will proceed as usual.

addvar `lpcontrol actvarlim` integer `;`

Limits the maximum number of variables which can be activated in any one execution of the variable activation phase. A value of 0 (the default) means that no limit is enforced.

check `lpcontrol check` integer `;`

$1 \leq \text{factor}/2 \leq \infty$

The nominal interval between accuracy checks, expressed in terms of the number of pivots which actually change the basis.

Accuracy checks attempt to detect the accumulation of numerical inaccuracy, and DYLP will perform a check earlier if it suspects numerical problems. While there's no enforced upper limit on the number of pivots between accuracy checks, in practice an accuracy check is performed each time the basis is factored during simplex phases.

coldvars `lpcontrol coldvars` integer `;`

$0 \leq 5000 \leq 100000$.

When the number of active variables in the constraint system on a cold start exceeds coldvars, and the client has not requested that DYLP work with the full constraint system, DYLP will attempt to deactivate variables before beginning simplex iterations.

The upper limit is soft; DYLP will issue a warning if a higher value is requested, but will not enforce the limit.

con actlvl, actlim, deactlvl

con.actlvl `lpcontrol actconlvl` integer ;

Specifies the constraint activation strategy. There are two levels:

0 (strict) Activate only constraints which are strictly violated.

1 (tight) Activate constraints which are tight or strictly violated.

con.actlim `lpcontrol actconlim` integer ;

Limits the maximum number of constraints which can be activated in any one execution of the constraint activation phase. A value of 0 (the default) means that no limit is enforced.

con.deactlvl `lpcontrol deactconlvl` [`normal`|`aggressive`|`fanatic`] ;

Specifies the constraint deactivation strategy. There are three levels:

0 (`normal`) Deactivate only inequalities which are strictly loose (*i.e.*, the associated slack is basic and not at bound).

1 (`aggressive`) (default) Deactivate loose inequalities and tight inequalities whose associated dual variable is zero.

2 (`fanatic`) Deactivate loose inequalities and any tight constraint (inequality or equality) whose associated dual variable is zero.

copyorigsys `lpcontrol forcecopy` boolean ;

If set to true, DYLP will always make a local copy of the original system. By default, a local copy is made only when necessary.

DYLP needs access to a copy of the original constraint system in order to scan it for constraints or variables that should be added. Normally this access is read-only, and DYLP uses the constraint system supplied as a parameter. When scaling is needed, DYLP makes a local copy of the original constraint system, applies scaling, and uses the scaled local copy as the original constraint system.

degen `lpcontrol antidegen` boolean ;

If set to false, DYLP will not use the perturbation-based anti-degeneracy algorithm described in §5. The default is to use perturbation-based anti-degeneracy.

degenlite `lpcontrol degenlite`
[`pivotabort`|`pivot`|`alignobj`| `alignedge`|`perpobj`|`perpedge`] ;

This option specifies the tie-breaking strategy used for choosing between candidates with equal deltas when selecting the leaving primal or dual variable, as described in §6. The options are:

0 (`pivotabort`) Break ties using the magnitude of the pivot coefficient, and abort the search at the first basic variable which gives a delta of zero.

1 (`pivot`) (default) Break ties using the magnitude of the pivot coefficient, scanning all basic variables.

2 (`alignobj`) Break ties by choosing the leaving variable which will make tight the hyperplane most closely aligned with the normal of the objective function (*i.e.*, the normal most nearly lies in the hyperplane).

3 (`alignedge`) Break ties by choosing the leaving variable which will make tight the hyperplane most closely aligned with the direction of motion specified by the entering variable (*i.e.*, the edge most nearly lies in the hyperplane).

4 (`perpobj`) Break ties by choosing the leaving variable which will make tight the hyperplane most nearly perpendicular to the normal of the objective function (*i.e.*, the hyperplane most nearly blocks motion in the direction of the normal of the objective)

5 (`perpedge`) Break ties by choosing the leaving variable which will make tight the hyperplane most nearly perpendicular to the direction of motion specified by the entering variable (*i.e.*, the hyperplane most nearly blocks motion in the direction of the edge).

degenpivlim `lpcontrol degenpivs` boolean `;`

$1 \leq 1 \leq \infty$

Limits the number of consecutive degenerate pivots which are required to trigger the perturbation-based anti-degeneracy algorithm. A perturbed subproblem is formed when the number of consecutive degenerate pivots exceeds degenpivlim. The current default of 1 is very aggressive.

dpsel: strat, flex, allownopiv

`lpcontrol dualmultipiv` integer `;`

There are four dual pivoting strategies accessible from the `dualmultipiv` command, specified by the following integer codes:

0 standard dual pivoting (*vid.* §12.5)

1 generalised dual pivoting (*vid.* §12.6); pivot chosen for maximum dual objective improvement

2 generalised dual pivoting; pivot chosen to mimimise the maximum infeasibility over primal variables

3 generalised dual pivoting; pivot chosen to minimise the maximum infeasibility over primal variables only if the infeasibility can be reduced; otherwise the pivot is chosen for maximum dual objective improvement

The pivoting strategy currently in use is held in dpsel.strat.

Two additional values are used to control generalised dual pivoting; these can only be changed under program control. dpsel.flex defaults to true, allowing DYLP to move between strategies 1 and 3. If the client specifies a pivoting strategy using the `dualmultipiv` command, dpsel.flex is set to false. dpsel.allownopiv controls whether DYLP will consider a generalised dual 'pivot' which consists of a sequence of variable flips without a final pivot. Computational experience says that this is very prone to cycling and dpsel.allownopiv is set to false by default.

The default initial setting for the dual pivoting options is dpsel.strat = 1, dpsel.flex = true, and dpsel.allownopiv = false.

dualadd `lpcontrol dualacttype` integer `;`

This option controls the amount of effort that DYLP will expend attempting to add variables (dual constraints) to bound a constraint system which is dual unbounded (*vid.* §14.2).

0 Variable activation is not attempted.

1 Type 1 variables are activated. These are variables which could potentially bound the dual problem and which will be dual feasible if activated and placed in the nonbasic partition. Multiple variables of this type can be activated simultaneously.

2 Type 2 variables will be activated if there are no type 1 variables. Type 2 variables are variables which would be dual infeasible if placed in the nonbasic partition, but which can be activated and immediately pivoted into the basis to regain dual feasibility. Only one variable of this type can be activated at a time, so this level is computationally expensive.

3 (default) Type 3 variables will be activated if there are no type 1 or type 2 variables. Type 3 variables are variables which can be activated and placed in the nonbasic partition with a bound-to-bound pivot.

If the limits placed on dual variable activation do not allow the dual to be bounded DYLP will revert to primal simplex. Allowing up to type 3 activations by default is somewhat risky; limiting activations to type 1 would be a more conservative choice.

factor `lpcontrol factor` integer `;`

$1 \leq 50 \leq 100$

The nominal interval for refactoring the basis, in terms of the number of pivots which actually change the basis.

Put another way, factor limits the total number of eta matrices in the multiplicative representation of the basis. As eta matrices accumulate, the work required to perform multiplication by the basis inverse increases, numerical inaccuracy increases, and the data structure grows (*vid.* §16.3). This parameter attempts to balance these considerations against the work required to refactor the basis. DYLP will refactor earlier if it suspects numerical problems.

The upper limit is soft; DYLP will issue a warning if a higher value is requested, but will not enforce the limit.

finpurge vars, cons

`lpcontrol final purge` *purge-spec*-LIST′ `;`
purge-spec ::= [`variables` | `constraints`] boolean

Specifies whether DYLP should perform a final round of constraint and/or variable deactivation when the problem has been solved to optimality. By default, DYLP will perform a final round of constraint deactivation and a final round of variable deactivation before it returns.

This application of constraint and/or variable deactivation is *not* suppressed by the fullsys option.

forcecold `lpcontrol cold` boolean `;`

When set to true, this option will force DYLP to perform a cold start. forcecold dominates forcewarm. The absence of forcecold and forcewarm allows a hot start.

forcewarm `lpcontrol warm` boolean `;`

When set to true, this option will force DYLP to perform a warm start. The absence of forcecold and forcewarm allows a hot start.

fullsys `lpcontrol fullsys` boolean `;`

When set to true, fullsys forces the use of the full constraint system at all times. DYLP will load the entire constraint system at startup and no constraint or variable activation or deactivation will be performed.

In the context of a branch-and-bound MIP code, where the bulk of the LPs are reoptimisations from a known basis, the use of dynamic simplex can save considerable work. To solve an LP once from scratch, or to solve the initial LP relaxation in a branch-and-bound context, use of the full system is usually (but not always) more efficient.

groom `lpcontrol groom` [`silent`|`warn`|`abort`] `;`

Specifies the action taken when DYLP detects a nontrivial change in the status of a variable when it performs a check following refactoring. The possible values are

0 (`silent`) Do nothing.

1 (`warn`) (default) Issue a warning message.

2 (`abort`) Issue an error message and force an abort.

The working assumption is that refactoring the basis removed accumulated numerical inaccuracy, causing the change in the status of the variable.

heroics: d2p, p2d

These parameters control whether DYLP will attempt difficult deactivations when trying to force a transition to dual or primal feasibility.

d2p If true, DYLP will attempt to deactivate primal infeasible basic architectural variables when trying to force primal feasibility.

p2d If true, DYLP will attempt to deactivate tight constraints (*i.e.*, nonbasic logicals) when trying to force dual feasibility.

Both of these default to false. Computational experience says that setting them to true is not useful. They can be adjusted only under program control.

idlelim `lpcontrol idle` integer `;`

$0 \leq 1000 \leq 2 * (\text{concnt} + \text{archvcnt}) \leq 50000 \leq 2^{\text{sizeof(int)}-3}$

The limit on the number of pivots allowed without an improvement in the value of the objective function.

A pivot in which the change in the objective function value is less than dy_tols.dchk is defined to be an idle pivot. Too many consecutive idle pivots are taken as an indication that the LP has stalled and may be cycling. If the number of pivots without change in the objective exceeds idlelim, DYLP aborts and returns lpSTALLED. Left to its own devices, DYLP will enforce the inner limits of $1000 \leq \text{idlelim} \leq 50000$; the client can explicitly specify any value within the outer limits.

initbasis `lpcontrol coldbasis` [`slack`|`logical`|`architectural`] `;`

This parameter specifies the type of initial basis constructed for a cold start, as described in §10.1.

1 (`logical`)	(default) Prefer slack, then artificial, variables for basic variables. Architectural variables will not be used.
1 (`slack`)	Prefer slack, then architectural, variables for basic variables. Artificial variables will be used if absolutely necessary.
2 (`architectural`)	Prefer architectural, then slack, variables for basic variables. Artificial variables will be used if absolutely necessary.

initcons: frac, i1lopen, i1l, i1uopen, i1u, i2lopen, i2l, i2uopen, i2u

`lpcontrol load` [*load-fraction*] *interval*-LIST′ `;`
load-fraction ::= float
interval ::= *open-delim ub lb close-delim*
ub ::= float
lb ::= float
open-delim ::= `(` | `[`
close-delim ::= `)` | `]`

These parameters control the loading of a partial constraint system during a cold start. As described in §10.1, constraints are ranked by the angle formed by the constraint normal and the objective normal, and a specified fraction of one or two angular intervals is loaded.

The parameter frac specifies what fraction of the inequalities in the specified intervals will be loaded. The parameters i1l and i1u specify the upper and lower bounds of one interval. If i1lopen is true, the lower boundary is open; if i1uopen is true, the upper boundary is open. The parameters i2l, i2u, i2lopen, and i2uopen can be used to specify an optional second interval.

A few examples will make the usage clear. By default, DYLP loads 50% of all inequalities, with the exception of inequalities which form an angle of 90° with the objective. This is specified as

```
lpcontrol load .5 [180 90) (90 0] ;
```

To load 75% of the inequalities with angles between 100° and 80°, inclusive, the specification would be

```
lpcontrol load .75 [100 80] ;
```

Loading the complete constraint system with the specification

```
lpcontrol load 1.0 [180 0] ;
```

is *not* equivalent to asking DYLP to always use the full constraint system (*cf.* fullsys). It will look pretty much the same from the outside, but DYLP will spend time internally performing scans related to constraint and variable activation and deactivation.

iterlim `lpcontrol iters` integer `;`

$0 \leq 10000 \leq 5 * (\text{concnt} + \text{archvcnt}) \leq 100000 \leq 2^{\text{sizeof(int)}-3}$

The pivot limit for each occurrence of a simplex phase (primal phases I and II and dual phase II). The overall pivot limit, cumulative over all occurrences of all phases, is $3 * \text{iterlim}$. If either the per phase or total limit is exceeded, DYLP terminates the problem and returns lpITERLIM. Left to its own devices, DYLP will enforce the inner limits of $10000 \leq \text{iterlim} \leq 100000$; the client can explicitly specify any value within the outer limits.

patch `lpcontrol patch` boolean `;`

If set to false, DYLP is forbidden from patching a singular basis. By default, DYLP will patch a singular basis and keep going. You really don't want to set this to false.

ppsel `lpcontrol primmultipiv` integer ;

There are two primal pivoting strategies accessible from the `primmultipiv` command, specified by the following integer codes:

0 standard primal pivoting (*vid.* §13.6)

1 (default) extended primal pivoting (*vid.* §13.7)

The pivoting strategy currently in use is held in ppsel.strat.

print `lpprint` *what* integer ;
what ::= `basis`|`conmgmt`|`crash`|`degen`|`dual`| `major`|`phase1`|`phase2`|`pivoting`|
`pivreject`|`pricing`|`scaling`|`setup`|`varmgmt`

The print options control the amount of output which DYLP produces as it runs. This can be varied from absolutely nothing to copious output useful only during detailed debugging. Printing options are covered in detail in §18, which describes debugging options and capabilities. If DYLP is compiled with the compile-time constant NDEBUG defined, virtually all informational printing is removed.

scaling `lpcontrol scaling` integer ;

Specifies how DYLP should scale the constraint system (§9).

0 DYLP is not allowed to apply scaling.

1 DYLP should use scaling vectors attached to the constraint system.

2 (default) DYLP should evaluate the constraint system and apply scaling if necessary.

scan `lpcontrol scan` integer ;

$200 \leq$ archvcnt/2 ≤ 1000.

Specifies the minimum number of columns which will be scanned in primal simplex to select a new candidate entering variable. This parameter applies only when dy_primalin is called to select the entering variable (*vid.* §13.5).

usedual `lpcontrol usedual` boolean ;

When set to false, this option prevents DYLP from using dual simplex. By default, DYLP will use dual simplex when possible.

16.11 DYLP Tolerances

DYLP has a number of numeric tolerances and related control information which are used in equality and accuracy checks and associated algorithms which attempt to control the accumulation of numerical accuracy. Each is described briefly below; again, the reader is encouraged to consult dylp.h for details.

Several of the tolerances described below are dynamically adjusted by DYLP in response to its assessment of the numerical stability of the current basis. As a general rule, tread carefully when overriding DYLP's defaults, and please take the time to read the code comments and consider the interrelationships between the tolerances.

bogus `lpcontrol bogus` double `;`

Default: 1.0

The 'bogus number' tolerance. Values such that zero $< |x| \leq$ zero $*$ bogus are considered likely to be the result of accumulated numerical inaccuracy, rather than legitimate values. Pivot coefficients and primal variable values within this range will trigger refactoring of the basis. For dual variables, the same test is applied, using the dual zero tolerance (cost). The default value is 1.0.

Experience seems to show that for the majority of problems increasing this value will cause the basis to be refactored more often and will not improve performance or accuracy. It's better to rely on DYLP's accuracy checks to determine if the basis should be refactored before the normal refactor interval has passed. Increasing bogus may be useful if scaling is disabled, or if factor has been set to a very large value.

cost `lpcontrol costz` double `;`

Default: 1.0×10^{-11}

The zero tolerance applied to values associated with the dual problem (dual variables and reduced costs).

This tolerance may be tightened if DYLP scales the constraint system for numerical stability. Let $\psi = ((\max_{ij}|a_{ij}|)/(\min_{ij}|a_{ij}|))^{1/2}$. Let ψ_u be the value calculated for the unscaled matrix A and ψ_s be the value calculated for the scaled matrix $\breve{A}$. Let $s = \max(0, \lfloor \log \psi_u/\psi_s + .5\rfloor - 2$. The dual zero tolerance will be tightened by 10^{-s} (*i.e.*, cost = cost $\times\, 10^{-s}$). In english, if scaling really did make a difference, so that the scaled matrix is significantly more stable than the unscaled matrix, DYLP should be extra careful about accuracy so that the scaled solution is still a solution after unscaling.

dchk `lpcontrol dchk` double `;`

Default: $1.0 \times 10^{s-4}$, where $s = \max(0, \lfloor \log \text{archccnt} + .5\rfloor - 2$

The dual accuracy check tolerance, as described in §8. The adjustment by s progressively loosens the accuracy check tolerance for systems with more than $10^{2.5} \approx 300$ dual variables. In english, when there are many dual variables, accumulating numerical inaccuracy warrants some relaxation of the accuracy check tolerance. This adjustment is made in dy_checkdefaults.

dfeas

The dual feasibility check tolerance, dynamically calculated using cost as the base value, as described in §8.

dfeas_scale `lpcontrol dfeas` double `;`

Default: $1.0 \times 10^{s+2}$, where $s = \max(0, \lfloor \log \text{archccnt} + .5\rfloor - 2$

Decoupling multiplier for scaling dfeas. This multiplier may be increased if the constraint system contains many dual variables or if the constraint system is scaled.

The adjustment for a large number of dual variables is the same adjustment applied for dchk.

The adjustment for matrix scaling follows the adjustment described for cost. Using the definitions for ψ_u and ψ_s given for cost, $s = \max(0, \lfloor \log \psi_u/\psi_s + .5\rfloor - 1$ and dfeas_scale will be increased by 10^s. In english, the separation between the dual zero tolerance and the dual feasibility tolerance is increased to compensate for tightening the dual zero tolerance.

inf `lpcontrol infinity` [`IEEE`|`DBL_MAX`|double] `;`

Infinity. DYLP can work with an infinite or finite infinity.

Default: HUGE_VAL

HUGE_VAL will be IEEE 754 infinity on most modern systems.

Many numerical programs still use that mathematical oxymoron, a finite infinity. Most commonly, this will be the value defined for the ANSI C symbol float.h:DBL_MAX, the maximum representable value for type double. Finite and infinite infinity do not play well together. If DYLP is being used by a client program which uses a finite infinity, set inf to the client's value of infinity.

pchk `lpcontrol pchk` double `;`

Default: $1.0 \times 10^{s-5}$, where $s = \max(0, \lfloor \log \text{archvcnt} + .5 \rfloor - 2$

The primal accuracy check tolerance, as described in §8. The adjustment by s progressively loosens the accuracy check tolerance for systems with more than $10^{2.5} \approx 300$ variables. In english, when there are many variables, accumulating numerical inaccuracy warrants some relaxation of the accuracy check tolerance. This adjustment is made in dy_checkdefaults.

pfeas

The primal feasibility check tolerance, dynamically calculated using zero as the base value, as described in §8.

pfeas_scale `lpcontrol pfeas` double `;`

Default: $1.0 \times 10^{s+2}$, where $s = \max(0, \lfloor \log \text{archvcnt} + .5 \rfloor - 2$

A decoupling multiplier used to adjust the separation of pfeas and zero as described in §8. This multiplier may be increased if the constraint system contains many variables or if the constraint system is scaled.

The adjustment for a large number of variables, specified with the default value, is the same adjustment applied for pchk. In english, when there are many variables, accumulating numerical inaccuracy warrants some relaxation of the feasibility tolerance.

The adjustment for matrix scaling follows the adjustment described for zero. Using the definitions for ψ_u and ψ_s given for zero, $s = \max(0, \lfloor \log \psi_u/\psi_s + .5 \rfloor - 1$ and pfeas_scale will be increased by 10^s. In english, the separation between the zero tolerance and the feasibility tolerance is increased to compensate for tightening the zero tolerance.

pivot `lpcontrol pivot` double `;`

Default: $1.0 \times 10^{-}5$

The pivot selection multiplier. A pivot coefficient $\overline{a}_{ij}$ will be accepted as numerically stable in the primal algorithm if $|\overline{a}_{ij}| \geq (\text{pivot})(\text{piv\_tol})\|\overline{a}_j\|_1$, where piv_tol is the stable pivot tolerance used during factoring in GLPK. In the dual algorithm, the 1-norm is calculated over the pivot row $\overline{a}_i$.

The pivot selection multiplier may be reduced if DYLP finds itself at an extreme point where all potential pivots x_i, x_j have been rejected because the pivot coefficients $\overline{a}_{ij}$ were judged numerically unstable (*vid.* §11.2).

In english, if pivot were set to 1, the pivot coefficient $\overline{a}_{ij}$ for every simplex pivot would have to satisfy the same stability criterion that the GLPK basis package applies when factoring the basis. This would be overly restrictive, however — when executing simplex pivots, DYLP needs to choose the pivot row and column to maximise progress toward an optimal extreme point. Some compromise is necessary; the value of pivot controls the balance between numerical stability and progress toward an optimal solution. When DYLP finds itself in a difficult spot, it will tilt the balance in order to make progress toward optimality.

purge `lpcontrol purgecon` double `;`

Default: 1.0×10^{-4}

The required percentage change in the value of the objective function before constraint or variable deactivation is allowed. This should be strictly greater than zero in order to minimise the possibility of a cycle involving activation/deactivation of constraints or variables.

purgevar `lpcontrol purgevar` double `;`

Default: .5

Used to calculate the variable deactivation threshold as a percentage of the maximum unfavourable reduced costs, as described in §14.3.

reframe `lpcontrol reframe` double `;`

Default: .1

The percentage error in the updated column or row norms which is required to trigger a reset of the PSE reference frame or the DSE row norms, respectively. A relatively large error can be tolerated here. The consequence of inaccuracy, a chance of a suboptimal choice of primal entering or dual leaving variable, is not too serious. In contrast, for the dual the computational cost of recalculating the basis inverse row norms $\|\beta_k\|$ is high. For the primal, all column norms are reset to 1, effectively reverting to unscaled ('Dantzig') pricing.

swing `lpcontrol swing` double `;`

Default: 1.0×10^{15}

This tolerance is used to detect excessive change in the values of the primal variables. The magnitude of the value prior to a pivot is compared to the magnitude after the pivot. If the ratio exceeds the value of swing, the simplex phase will abort and DYLP will attempt to bound the primal swing (*vid.* §11.2).

toobig

Default: 1.0×10^{30}.

This value is used to control changes in the dual multipivot strategy. The breakpoints are currently hardcoded in dy_dualmultipivot:dualmultiin (which see).

zero `lpcontrol zero` double `;`

Default: 1.0×10^{-11}.

The zero tolerance. Values smaller than |zero| are set to a clean floating-point zero.

This tolerance may be tightened if DYLP scales the constraint matrix for numerical stability. Let $\psi = ((\max_{ij}|a_{ij}|)/(\min_{ij}|a_{ij}|))^{1/2}$. Let ψ_u be the value calculated for the unscaled matrix A and ψ_s be the value calculated for the scaled matrix $\breve{A}$. Let $s = \max(0, \lfloor \log \psi_u/\psi_s + .5 \rfloor - 2$. The zero tolerance will be tightened by 10^{-s} (*i.e.*, zero = zero $\times\, 10^{-s}$). In english, if scaling really did make a difference, so that the scaled matrix is significantly more stable than the unscaled matrix, DYLP should be extra careful about accuracy so that the scaled solution is still a solution after unscaling.

17 DYLP Statistics

DYLP will collect detailed statistics if the conditional compilation symbol DYLP_STATISTICS is defined. The available statistics are described briefly in the paragraphs which follow; for details on sub-fields, consult dylp.h. Routines in the file statistics.c provide initialisation (dy_initstats), printing (dy_dumpstats), and release of the data structure (dy_freestats).

angle: **max, min, hist**

Statistics on the angles of inequality constraints to the objective function. For constraint i, this is calculated as $\frac{180}{\pi}\cos^{-1}\frac{a_i c}{\|a_i\|\|c\|}$. The maximum and minimum angle is recorded, and a histogram in 5° increments with a dedicated 90° bin.

cons: **sze, angle, actcnt, deactcnt, init, fin**

Information about individual constraints: the angle of the constraint with the objective function, the number of times it's activated and deactivated, and booleans to indicate if the constraint is active in the initial and final active systems.

d2: **pivs, iters**

Total pivot and iteration counts for DYLP. The pivot count is the number of successful simplex pivots. The iteration count also includes pivot attempts which did not succeed for some reason (*e.g.*, a primal pivot in which the entering variable was eventually rejected because the pivot element was numerically unstable).

ddegen: **cnt, avgsiz, maxsiz, totpivs, avgpivs, maxpivs**

Statistics on the amount of time spent in restricted subproblems trying to escape dual degeneracy.

For each level (*i.e.*, each nested level of restricted subproblem), DYLP records the number of times this level was reached, the average and maximum number of variables involved in a degeneracy, the total and average number of pivots executed at this level, and the maximum number of pivots executed in any one subproblem at this level. The array is generously sized (by compile time constant) to accommodate a maximum of 25 levels.

dmulti: **flippable, cnt, cands, promote, nontrivial, evals, flips, pivrnks, maxrnk**

Statistics on the behaviour of the generalised dual pivoting algorithm. Each call to dualmultiin collects a list of candidate variables to enter the basis and sorts the list. This process may produce a unique candidate for entry, or it may leave a list of requiring further evaluation to determine the best sequence of flips and final pivot.

The flippable field records the number of flippable variables in the problem (*i.e.*, variables with finite lower and upper bounds). The cnt field records the total number of calls to dualmultiin, and nontrivial records the number of times the initial scan and sort did not identify a unique entering variable.

The remaining fields, with one exception, are totals. They record the number of candidates queued for evaluation, the number of times that a sane pivot was promoted over an unstable pivot, the number of columns transformed ($B^{-1}a_k$) for evaluation, the number of bound-to-bound flips, the rank in the sorted list of the variable selected to enter, and the maximum rank for a variable selected to enter.

factor: **cnt, prevpiv, avgpivs, maxpivs**

Statistics about basis factoring. The cnt field records the total number of times the basis was refactored. The avgpivs and maxpivs fields record the average and maximum number of pivots between basis refactoring.

infeas: **prevpiv, maxcnt, totpivs, maxpivs, chgcnt1, chgcnt2**

Statistics on the resolution of infeasibility during primal phase I.

The maximum number of infeasible variables is recorded, as well as the total pivots in phase I and the maximum number of pivots with no change in the number of infeasible variables. DYLP also counts the number of times that the number of infeasible variables changed without requiring recalculation of the reduced costs (chgcnt1), and the number of times when it did (chgcnt2). Specifically, if exactly one variable gains feasibility, and it leaves the basis as it does so, the reduced costs do not have to be recalculated.

p1: **pivs, iters**

Total pivot and iteration counts for primal phase 1 simplex.

p2: **pivs, iters**

Total pivot and iteration counts for primal phase 2 simplex.

pdegen: **cnt, avgsiz, maxsiz, totpivs, avgpivs, maxpivs**

Statistics on the amount of time spent in restricted subproblems trying to escape primal degeneracy. The content of individual fields is as for ddgen.

pivrej: **max, mad, sing, pivtol_red, min_pivtol, puntcall, puntret**

Statistics on the management of variables judged unsuitable for pivoting. Variables are queued on the rejected pivot list when a pivot attempt fails because the pivot element is numerically unstable or because the pivot produced a singular basis. During primal simplex, candidate entering variables are queued; during dual simplex, candidate leaving variables.

The max field records the maximum length of the rejected pivot list. The fields mad and singular record the number of variables queued for unstable pivots and singular basis, respectively.

The puntcall field records the number of times the routine dy_dealWithPunt was called in an attempt to remove variables from the rejected pivot list. The pivtol_red field records the number of times that the pivot selection multiplier was reduced in order to consider candidate variables previously rejected for numeric instability; min_pivtol is the minimum multiplier value used. The puntret field records the number of times dy_dealWithPunt was unable to remove any candidates from the rejected pivot list and therefore recommended termination of the current simplex phase.

pmulti: **cnt, cands, nontrivial, promote**

Statistics on the behaviour of the extended primal pivoting algorithm. Each call to primalmultiout collects a list of candidate variables to leave the basis. This process may produce a unique candidate to leave, or it may leave a list of candidates requiring further evaluation to determine the final pivot.

The cnt field records the total number of calls to primalmultiout, and nontrivial records the number of times the initial scan did not identify a unique leaving variable. The promote field records the number of times that a sane pivot was promoted over an unstable pivot,

tot: **pivs, iters**

Total pivot and iteration counts for the call to dylp.

vars: **sze, actcnt, deactcnt**

Information about individual variables: the number of times a variable is activated and deactivated.

18 DYLP Debugging Features

DYLP incorporates two types of debugging features: a controllable printing facility and paranoid checks. The printing facility is enabled when the symbol NDEBUG is not defined at compile time, and is intended to allow the generation of log information at whatever level of detail is desired by the user. The paranoid checks are enabled when the symbol PARANOIA is defined at compile time and are intended to provide significant (and expensive) cross-checks during code development.

18.1 Printing

The amount of output generated by DYLP can be varied from next to nothing to a level of detail intended only for detailed debugging. The paragraphs which follow briefly outline the capabilities; for specific output at a given print level, please refer to the file dylp.h.

basis Prints information related to management of the basis, including adjustments to suppress numerical instability and recover from singularity.

conmgmt Prints information on the management of constraints, including activation and deactivation, changes to primal and dual variables, and (at the highest level) a running commentary on all constraint and variable additions, deletions, and motions attributable to activation and deactivation of constraints.

crash Prints information regarding the generation of the initial basis, including factoring, the initial set of basic variables, and their values. For a cold start, information on the selection of the basic variables can be printed.

degen Prints information about degenerate pivots and restricted subproblem formation to deal with degeneracy.

dual Prints information about the execution of the dual simplex, with capabilities similar to phase1.

major Tracks the major state transitions of the dynamic simplex algorithm as DYLP solves an LP.

phase1 Prints information about the execution of phase I of the primal simplex. At the low end, messages are printed for extraordinary events — unboundedness, serious pivoting problems, *etc.* At a medium level, a one line message is printed summarising each pivot, as well as messages about routine but infrequent events — refactoring, accuracy checks, and various minor problems. At the highest level, all primal and dual variables are printed as they are recalculated for each pivot, along with detailed information about reduction of infeasibility and changes to the phase I objective function. This is *an enormous amount* of output for large problems.

phase2 Prints information about the execution of phase II of the primal simplex, with capabilities similar to phase1.

pivoting Prints information on the evaluation of candidates for the leaving primal or entering dual variable and details of the pivot column or row. At least one line per pivot; at the highest level, produces *a lot* of output.

pivreject Prints information on the operation of DYLP's pivot rejection mechanism.

pricing Prints information regarding the pricing of candidates for the entering primal or leaving dual variable. At any level above 1 you'll get *many* lines of output per pivot; that's *an enormous amount* of output for large problems.

scaling Prints information regarding numerical scaling of the constraint system.

setup Prints information regarding the loading and initialisation of an LP problem, including the constraints and variables which are activated and the angle of inequalities to the objective function.

varmgmt Prints information on the activation and deactivation of variables, much as conmgmt.

18.2 Paranoia

Because it is intended as a development code, DYLP incorporates a large number of sanity checks, enabled by defining the conditional compilation symbol PARANOIA. Many of these tests are cheap and simple — checks for null parameters, sensible constraint and variable counts, proper major phase, and range checks on indices. Others are more elaborate and expensive.

There are two dedicated subroutines which are used at several points to check the integrity of the current simplex point (basis, status, and primal variable values) and the constraint system:

- dy_chkstatus implements extensive checks to make sure that the status and value of a primal variable agree across multiple data structures and are appropriate for the current major phase.
- dy_chkdysys implements extensive checks to ensure that the active constraint system and associated data structures are correct and consistent.

There is another set of checks which track the numerical accuracy of calculations by performing an independent calculation of a quantity. These are of little use unless there is some reason to doubt the correctness of the calculation, hence the separate conditional compilation symbols.

- Checks on the accuracy of the calculations to produce unscaled rows of the basis inverse are controlled by the symbol CHECK_UNSCALED_BETAI.
- Checks on the accuracy of iterative updating for PSE column norms and DSE row norms are controlled by the conditional compilation symbols CHECK_PSE_UPDATES and CHECK_DSE_UPDATES. These checks calculate the norms directly for comparison with the updated values, and the computational expense is unacceptable unless there is specific reason to suspect an error.

References

[1] R. Bixby. Implementing the Simplex Method: The Initial Basis. *ORSA Journal on Computing*, 4(3):267–284, Summer 1992.

[2] COIN-OR (Common Infrastructure for Operations Research). Available at `http://www.coin-or.org`.

[3] J. Forrest and D. Goldfarb. Steepest-edge simplex algorithms for linear programming. *Mathematical Programming*, 57:341–374, 1992.

[4] L. Hafer. A Note on the Relationship of Primal and Dual Simplex. Technical Report SFU-CMPT TR 1998-21, School of Computing Science, Simon Fraser University, Burnaby, B.C., V5A 1S6, December 1998.

[5] L. Hafer. CONSYS: a dynamic constraint system. Technical Report SFU-CMPT TR 1998-22, School of Computing Science, Simon Fraser University, Burnaby, B.C., V5A 1S6, December 1998.

[6] A. Makhorin. GLPK (GNU Linear Programming Kit). Available at `http://www.gnu.org/software/glpk/glpk.html`.

[7] A. Makhorin. GLPK Linear Programming Kit: Implementation of the Revised Simplex Method. Glpk documentation, Moscow Aviation Institute, Moscow, Russia, February 2001.

[8] I. Maros. *Computational Techniques of the Simplex Method*. Kluwer Academic Publishers, Norwell, Massachusetts, 2003.

[9] M. Padberg. *Linear Optimization and Extensions*, volume 12 of *Algorithms and Combinatorics*. Springer-Verlag, New York, 1995.

[10] D. Ryan and M. Osborne. On the Solution of Highly Degenerate Linear Programmes. *Mathematical Programming*, 41:385–392, 1988.