Appendix B

Integer linear programming

This appendix introduces the **Simplex algorithm** for **linear programming**, with variables in $\mathbb{R}$, and the more difficult problem **integer linear programming** with variables in $\mathbb{N}$. The purpose is to explain the principles and the pseudo code of implementations of basic solvers for these problems, which are widely used in industry. We give no proofs and instead refer the reader to [67].

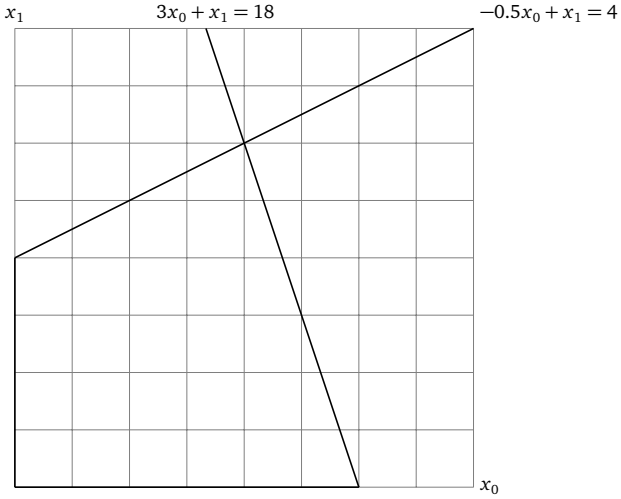

Figure B.1: A convex region with vertices $(0,0), (0,4), (4,6)$ and $(6,0)$.

Consider Figure B.1 with a region enclosed by the four lines $x_0 = 0$, $x_1 = 0$, $3x_0 + x_1 = 18$, and $-0.5x_0 + x_1 = 4$. The region defines a set of points in which we are looking for some point which maximizes (or minimizes) a linear function such as $z = x_0 + 2x_1$. This is an example of a linear program, which consists of

linear constraints on variables x_i and a linear objective function which should be maximized (or minimized). The linear program which results in the previous figure is

$$\begin{array}{lrcrcl} \max & z & = x_0 + 2x_1 & & & \\ & & & & & \\ & -0.5x_0 & + & x_1 & \leq & 4 \\ & 3x_0 & + & x_1 & \leq & 18. \end{array}$$

Here x_0 and x_1 are called **decision variables**. If some variables are in $\mathbb{R}$ and some others are $\mathbb{N}$, we have a **mixed integer linear program**. The most widely used algorithm for solving linear programs is the Simplex algorithm, invented in 1947 by George Dantzig after arriving late at a lecture at Berkeley and mistaking an open research problem for a home assignment. Although the Simplex algorithm is not in $\mathbb{P}$, it is very fast in practice, but exponential in the worst case. Solving an integer linear program is in general an NP-complete problem. A **branch-and-bound** algorithm introduced in Section B.4, on page 726, for solving integer linear programs makes use of the Simplex algorithm for solving linear programs as a subproblem.

The chapter is organized as follows. In Section B.1, on page 712, we introduce some mathematical aspects of linear programming to prepare for the explanation of the Simplex algorithm in Section B.2, on page 716.

B.1 Linear programs

A linear program consists of an objective function with n variables $\sum c_i x_i$ which either should be minimized or maximized and m linear constraints on x_i. In addition, we require $x_i \geq 0$. With $\mathbf{c} = (c_0, c_1, ..., c_{n-1})$, $\mathbf{b} = (b_0, b_1, ..., b_{m-1})$, and $\mathbf{A}$ an (m, n) matrix, we can write a linear program as

$$\begin{array}{lrcl} \max & z = c_0x_0 + c_1x_1 + ... + c_{n-1}x_{n-1} & & \\ & & & \\ & a_{0,0}x_0 + a_{0,1}x_1 + ... + a_{0,n-1} & \leq & b_0 \\ & a_{1,0}x_0 + a_{1,1}x_1 + ... + a_{1,n-1} & \leq & b_1 \\ & ... & & \\ & a_{m-1,0}x_0 + a_{m-1,1}x_1 + ... + a_{m-1,n-1} & \leq & b_{m-1} \\ & & & \\ & x_0, x_1, ..., x_{n-1} & \geq & 0 \end{array} \tag{B.1}$$

or simpler as

$$\begin{array}{lrcl} \max & z = \mathbf{cx} & & \\ & \mathbf{Ax} & \leq & \mathbf{b} \\ & \mathbf{x} & \geq & \mathbf{0}. \end{array} \tag{B.2}$$

The non-negativity constraints are not counted in the m other constraints. Each constraint defines a halfplane in n dimensions and the intersection of these halfplanes defines the **feasible region**, $\mathbf{P}$, with **feasible solutions** $\mathbf{x} \in \mathbf{P}$. The feasible region is convex, and a point where halfplanes intersect is called a **vertex**. A linear program is either **infeasible** when $\mathbf{P}$ is empty, **unbounded** when no finite solution exists, or **feasible**, in which case we search for an optimal solution $\mathbf{x}^* \in \mathbf{P}$ which maximizes z. There may exist more than one optimal solution. We denote by $z(\mathbf{x})$ the value of the objective function z at point $\mathbf{x}$. A solution $\mathbf{x}$ is **local optimum** for $z(\mathbf{x})$ if there is an $\epsilon > 0$ such that $z(\mathbf{x}) \geq z(\mathbf{y})$ for all $\mathbf{y} \in \mathbf{P}$ with $||x - y|| \leq \epsilon$.

We give the following theorem without a proof.

Theorem B.1 *A local optimum of a linear program is also a global optimum.*

Having seen that a local optimum is also a global optimum, we will next prove another convenient fact about linear programs which limits where we need to search for an optimal solution.

Theorem B.2 *For a bounded feasible linear program with feasible region P, at least one vertex is an optimal solution.*

Now knowing that the optimal solution can be found in a vertex, we will next see how the linear constraints and the vertices are related. So far we have expressed linear programs with inequalities, which is called **standard form**. It is, however, more convenient to work with equalities and therefore for each of the m constraints, we add another variable and rewrite the constraints as equalities, which is called **slack form**.

$$
\begin{array}{lrcll}
max & \mathbf{cx} & & & \\
 & x_{n+0} & = & b_0 - \sum_{j=0}^{n-1} a_{0,j} x_j & \\
 & x_{n+1} & = & b_1 - \sum_{j=0}^{n-1} a_{1,j} x_j & \qquad \text{(B.3)} \\
 & & \ldots & & \\
 & x_{n+m-1} & = & b_{m-1} - \sum_{j=0}^{n-1} a_{m-1,j} x_j & \\
 & x_i & \geq & 0 \qquad\qquad 0 \leq i \leq n+m-1 &
\end{array}
$$

The variables on the left hand side are called **basic variables** and occur only once, i.e. neither in any sum on the right hand side, nor in the objective function. The other variables are called **nonbasic variables**. There are m basic and n nonbasic variables. To be clear, the definition of a basic variable is that it is on

the left hand side of an equation, and thus the definition is not that the variable's index i satisfies $n \leq i \leq n+m-1$, although that happens to be the case initially.

We will also add a constant term y, initially zero, to the objective function. We will always represent a linear program in this form, but we will perform algebraic manipulations so that a particular variable x_k sometimes is a basic and and at other times is a nonbasic variable, depending on whether it to the left or right of the $=$. A variable is identified by its index and when we have found an optimal solution, it is the values of $x_0, x_1, ..., x_{n-1}$ which are of interest.

An important observation is that if each coefficient c_i in the objective function is negative, we have found an optimal solution, with value y, each nonbasic variable set to zero, and each basic variable set to the corresponding b_i constant (the a_{ij} coefficients can be ignored since they are multiplied with nonbasic variables which are zero).

A strategy to find an optimal solution therefore is to rewrite the problem until all $c_i < 0$. So, we find any nonbasic variable x_k with a positive coefficient c_i, and rewrite the equations so that x_k becomes basic and a basic variable x_j becomes nonbasic. Recall, the index k of the variable x_k is that variable's "identity" and it is located in a column i of the right hand side, i.e., both in the objective function and the constraints. This nonbasic variable x_k is called an **entering basic variable** and the basic variable x_j is called a **leaving basic variable**, because x_j will become nonbasic. When we rewrite the equations, the index of a variable is never changed. There are always n nonbasic and m basic variables. Assume x_k is the nonbasic and x_j is the basic and we want them to switch roles. To do so, we simply take the row with x_j and rearrange it so that x_k becomes the left hand side variable. We also need to rewrite the other rows and the objective function so they no longer contain x_k. During the rewrite, the values of the **A** matrix, the **b** and **c** vectors and the constant term y change, but not the value of the objective function. This rewriting is called a **pivot operation**.

Example B.1.1 Consider again the linear program in Figure B.1. In this example we will illustrate how it can be solved. We start with the linear program on standard form:

$$\begin{aligned} \max \quad z \quad &= x_0 + 2x_1 \\ \\ -0.5x_0 \quad &+ \quad x_1 \quad \leq \quad 4 \\ 3x_0 \quad &+ \quad x_1 \quad \leq \quad 18 \end{aligned}$$

and then introduce two new variables, one for each linear constraint, and write it on slack form:

$$\max \quad z \quad = x_0 + 2x_1$$

$$\begin{aligned} x_2 &= 4 - (-0.5x_0 + x_1) \\ x_3 &= 18 - (3x_0 + x_1). \end{aligned}$$

The strategy is to rewrite the problem until all coefficients in the objective function are negative and then set the nonbasic variables to zero. The optimal value will then be the constant term in the objective function. The basic variables are given by their equation which will be the corresponding b_i. We select a variable with a positive c_i coefficient. Let us take x_0, which therefore becomes the entering basic variable. Since c_0 is positive, we want in principle to increase the value of x_0 as much as possible as that increases z. As mentioned, we will rewrite the problem in order to have only negative coefficients of the nonbasic variables since we then have found the optimal value of z. In order to achieve that, we want to increase the value of x_0 as much as possible. The basic variables limit how much x_0 can be increased, and it is x_3 which is more restrictive since increasing x_0 is limited by $4-0.5x_0 \geq 0$ for x_2 and $18-3x_0 \geq 0$ for x_3. Therefore we select x_3 as the leaving basic variable, and after rewriting the problem it becomes:

$$\max \quad z = -0.333x_3 + 1.667x_1 + 6$$

$$\begin{aligned} x_2 &= 7 - (0.167x_3 + 1.167x_1) \\ x_0 &= 6 - (0.333x_3 + 0.333x_1) \end{aligned}$$

Since we set the nonbasic variables to zero, we can make the observation that the initial problem with x_0 and x_1 as nonbasic had a value of the objective function $z = z_0 + 2x_1 = 0$, which corresponds to vertex $(0, 0)$ of the feasible region. In the current problem with nonbasic variables x_3 and x_1 set to zero and $x_0 = b_1 = 6$, and $z = x_0 + 2x_1 = 6$, and the values of x_0 and x_1 correspond to vertex $(6, 0)$ in the feasible region.

In this problem only $c_1 > 0$ and x_1 becomes entering basic variable. At this point x_2 is most constrained and becomes leaving basic variable, and with the rewritten problem it is solved.

$$\max \quad z = -0.571x_3 - 1.429x_2 + 16$$

$$\begin{aligned} x_1 &= 6 - (0.143x_3 + 0.857x_2) \\ x_0 &= 4 - (0.286x_3 - 0.286x_2) \end{aligned}$$

We see that $x_0 = 4$ and $x_1 = 6$, i.e. vertex $(4, 6)$ gives $z = 16$. In this example, both x_0 and x_1 happen to become basic variables in the end but this is not in general the case. ■

The variable assignment of zero to each nonbasic variable and the corresponding element of the $\mathbf{b}$ vector to each basic variable is called a **basic solution**, and a **basic feasible solution** if it is feasible (i.e. all constraints satisfied). In our

example, each basic feasible solution corresponded to one vertex and it can be shown that this is so in general. See for instance [64]. In the next section, we will present the Simplex algorithm.

B.2 The Simplex algorithm

The main idea of the Simplex algorithm is to improve the value of the objective function by moving from a vertex to another vertex with a higher value of z. Consider a linear program P_0 in which $\mathbf{b} \geq \mathbf{0}$. We can then start with setting all original decision variables (which from the beginning all are nonbasic) to zero, since then each basic variable becomes non-negative, and we have a basic feasible solution and can start the algorithm. In a linear program where $\mathbf{x} = \mathbf{0}$ is not in the feasible region, at least one b_i is negative. In this case we need to find another vertex v to start in. To find v, we create a new linear program P_1 as follows. Starting with $\mathbf{A}$ and $\mathbf{b}$ from P_0 we create a new nonbasic variable x_{n+m}. Instead of the objective function from P_0 we use $z_1 = -x_{n+m}$. From each constraint in the standard form of P_1, we subtract x_{n+m} giving each constraint i the form:

$$a_{i,0}x_0 + a_{i,1}x_1 + ... + a_{i,n-1} - x_{n+m} \quad \leq \quad b_i$$

If P_0 has a feasible solution $\mathbf{x}$, then P_1 will as well and with optimal value zero, i.e. $x_{n+m} = 0$, since the $\mathbf{x}$ from P_0 concatenated with $x_{n+m} = 0$ then is a feasible solution for P_1 which maximizes its objective function. If P_1 has optimal value 0 then $x_{n+m} = 0$ and by removing x_{m+n} from this solution, P_0 will have a feasible solution. Two interesting questions then are:

- can we easily find a feasible solution as start vertex for P_1?
- if P_1 is feasible, can we use its solution to find a start vertex v for P_0?

The answer to the first question is yes, and to the second that we continue with P_1 but with the objective function of P_0.

By doing a pivot on P_1 with x_{m+n} as entering basic variable and x_k with smallest b_k (i.e., most negative) as leaving basic variable, we get $\mathbf{b} \geq \mathbf{0}$, so P_1 can be solved using the Simplex algorithm, and if the value assigned to x_{m+n} is zero (or, in practical terms, $|x_{m+n}| < \epsilon$) we know both P_0 and P_1 are feasible. If x_{m+n} is a basic variable in the solution of P_1, we perform one additional pivot to make x_{m+n} nonbasic, so that x_{m+n} can be removed from P_1 (since its value is zero). We then take the objective function of P_0 and use it in P_1 by going through each of the n nonbasic variables in P_0, i.e. x_k for $0 \leq k < n$. We check if x_k is nonbasic or basic in the solution of P_1. If it is nonbasic, we can use the coefficient of x_k of the the objective function of P_0 directly while if it is basic, we need to take into account the coefficients of $\mathbf{A}$ of the solution of P_1 in the row where x_k is a

basic variable. The details can be found in the function *initialize* in the pseudo code. Before translating the pseudo code of this and the *pivot* function to source code, we strongly recommend the reader to manually perform the calculations with pen and paper. Although the algorithm might appear to be recursive since *xsimplex* and *initialize* call each other, there is only one level of recursion: when *initialize* is called the second time by *xsimplex*, we have $\mathbf{b} \geq \mathbf{0}$, so it returns without calling *xsimplex* again.

Vertices which are neighbors have all except one basic variable in common. Therefore, when performing a pivot operation we move from one vertex to a neighboring vertex. With m basic and n nonbasic variables, there are $\binom{n+m}{m}$ ways to select m basic variables from all $m + n$ variables. There is a small risk of not making progress and instead cycles which can be discovered after the same number of pivot operations. If that happens, we can switch to select entering and leaving basic variables using Bland's rule which guarantees progress but is slower. It states that of the nonbasic variables with a positive coefficient the one with smallest index should be selected, and if there are multiple leaving basic variables with the same ratio, again the one with smallest index should be selected.

The pseudo code below was translated from the author's C implementation. The purpose of it is to be as trivial as possible without allocating additional memory. For instance, it assumes there is memory allocated for the extra decision variable needed when $\mathbf{x} = \mathbf{0}$ is not a feasible initial basic solution, and the order in which the statements in a pivot operation are performed can reuse memory allocated for $\mathbf{A}$, $\mathbf{b}$ and $\mathbf{c}$.

Due to finite precision of floating point variables, it is not sufficient to simply compare a number against zero to see if it is nonzero. We use $\epsilon = 10^{-6}$.

Algorithm B.1 The Simplex algorithm.

Note that one extra column is assumed to have been allocated, for x_{m+n}.

```
struct simplex_t {
    int       m;              /* Constraints. */
    int       n;              /* Decision variables. */
    int       var[n+m+1];     /* 0..n-1 are nonbasic. */
    double    a[m][n+1];      /* A. */
    double    b[m];           /* b. */
    double    x[n+1];         /* x. */
    double    c[n];           /* c. */
    double    y;              /* y. */
}
```

```
function init(s, m, n, a, b, c, x, y, var)
begin
    int i,k
    *s = (m,n,a,b,c,x,y,var) // assign each attribute
    if s.var = null then
        s.var = new int [m+n+1]
        for (i = 0; i < m+n; i = i + 1)
            s.var[i] = i
    for (k = 0, i = 1; i < m; i = i + 1)
        if b[i] < b[k] then
            k = i
    return k
end
```

```
function select_nonbasic(s)
begin
    int i
    for (i = 0; i < s.n; i = i + 1)
        if s.c[i] > ϵ then
            return i
    return -1
end
```

```
procedure prepare(s, k)
begin
    int     m = s.m
    int     n = s.n
    int     i
    // make room for x_{m+n} at s.var[n] by moving s.var[n..n+m-1] one
    // step to the right.
    for (i = m + n; i > n; i = i − 1)
        s.var[i] = s.var[i − 1]
    s.var[n] = m + n
    // add x_{m+n} to each constraint
    n = n + 1
    for (i = 0; i < m; i = i + 1)
        s.a[i][n − 1] ← −1
    s.x = new double [m + n]
    s.c = new double [n]
    s.c[n − 1] = −1
    s.n = n
    pivot(s, k, n-1)
end


function initial(s, m, n, a, b, c, x, y, var)
begin
    int     i,j,k
    double w
    k = init(s, m, n, a, b, c, x, y, var)
    if b[k] ≥ 0 then
        return 1 // feasible
    prepare(s,k)
    n = s.n
    s.y = xsimplex(m, n, s.a, s.b, s.c, s.x, 0, s.var,1)
    for (i = 0; i < m+n; i = i + 1) {
        if s.var[i] = m+n-1 then
            if |s.x[i]| > ϵ then
                delete s.x
                delete s.c
                return 0 // infeasible
            else
                break // This i will be used on the next page.
    }

    // The rest of this function is on the next page.
```

```
    if i ≥ n then
        // x_{n+m} is basic. find good nonbasic.
        for (j = k = 0; k < n; k = k + 1)
            if |s.a[i−n][k]| > |s.a[i−n][j]| then
                j = k
        pivot(s,i-n,j)
        i = j
    if i < n-1 then
        // x_{n+m} is nonbasic and not last. swap columns i and n-1
        k = s.var[i]; s.var[i] = s.var[n-1]; s.var[n-1] = k
        for (k = 0; k < m; k = k + 1)
            w = s.a[k][n-1]; s.a[k][n-1] = s.a[k][i]; s.a[k][i] = w
    else
        // x_{n+m} is nonbasic and last. forget it.
    delete s.c
    s.c = c
    s.y = y
    for (k = n-1; k < n+m-1; k = k + 1)
        s.var[k] = s.var[k+1]
    n = s.n = s.n - 1
    t = new double [n]
    for (k = 0; k < n; k = k + 1) {
        for (j = 0; j < n; j = j + 1)
            if k = s.var[j] then
                // x_k is nonbasic. add c_k
                t[j] = t[j] + s.c[k]
                goto next_k
        // x_k is basic.
        for (j = 0; j < m; j = j + 1)
            if s.var[n+j] = k then
                // x_k is at row j
                break
        s.y = s.y + s.c[k] * s.b[j]
        for (i = 0; i < n; i = i + 1)
            t[i] = t[i] - s.c[k] * s.a[j][i]
    next_k:;
    }
    for (i = 0; i < n; i = i + 1)
        s.c[i] = t[i]
    delete t and s.x
    return 1
end
```

```
procedure pivot(s, row, col)
begin
    auto    a = s.a
    auto    b = s.b
    auto    c = s.c
    int     m = s.m
    int     n = s.n
    int     i,j,t
    t = s.var[col]
    s.var[col] = s.var[n+row]
    s.var[n+row] = t
    s.y = s.y + c[col] * b[row] / a[row][col]
    for (i = 0; i < n; i = i + 1)
        if i ≠ col then
            c[i] = c[i] - c[col] * a[row][i] / a[row][col]
    c[col] = - c[col] / a[row][col]
    for (i = 0; i < m; i = i + 1)
        if i ≠ row then
            b[i] = b[i] - a[i][col] * b[row] / a[row][col]
    for (i = 0; i < m; i = i + 1)
        if i ≠ row then
            for (j = 0; j < n; j = j + 1)
                if j ≠ col then
                    a[i][j] = a[i][j] - a[i][col] * a[row][j] / a[row][col]
    for (i = 0; i < m; i = i + 1)
        if i ≠ row then
            a[i][col] = -a[i][col] / a[row][col]
    for (i = 0; i < n; i = i + 1)
        if i ≠ col then
            a[row][i] = a[row][i] / a[row][col]
    b[row] = b[row] / a[row][col]
    a[row][col] = 1 / a[row][col]
end
```

```
function xsimplex(m, n, a, b, c, x, y, var, h)
begin
    simplex_t   s
    int         i,row,col
    if !initial(&s, m, n, a, b, c, x, y, var) then
        delete s.var
        return NaN // not a number
    while ((col ← select_nonbasic(&s)) ≥ 0) {
        row ← -1
        for (i = 0; i < m; i = i + 1)
            if a[i][col] > ϵ  and
                (row < 0  or b[i] / a[i][col] < b[row] / a[row][col]) then
                row = i
        if row < 0 then
            delete s.var
            return ∞ // unbounded
        pivot(&s,row, col)
    }
    if h = 0 then
        for (i = 0; i < n; i = i + 1)
            if s.var[i] < n then
                x[s.var[i]] = 0
        for (i = 0; i < m; i = i + 1)
            if s.var[n+i] < n then
                x[s.var[n+i]] = s.b[i]
        delete s.var
    else
        for (i = 0; i < n; i = i + 1)
            x[i] = 0
        for (i = n; i < n+m; i = i + 1)
            x[i] = s.b[i-n]
    return s.y
end



function simplex(m, n, a, b, c, x, y)
begin
    return xsimplex(m,n,a,b,c,x,y,null,0)
end
```

B.3 Integer linear programming

Integer linear programming is an NP-complete problem in which the decision variables are integers, and therefore, no efficient algorithm for finding an optimal solution exists. Again, we have n decision variables and m constraints. If we **relax** the integer requirement, and try the Simplex algorithm and it happens to produce an integer solution, i.e. with each $x_i \in \mathbb{N}$, then that is the optimal solution, and we are ready. If it does not, let us denote the initial integer program p, the value returned by the Simplex algorithm for the relaxed problem $p.z$ and the variable assignment $p.x$. Rounding each non-integer $p.x_i$ does not work, but we still have use for the Simplex algorithm.

Suppose $p.x_k = u \notin \mathbb{N}$. We can then create two new problems, one with the additional constraint $x_k \leq \lfloor u \rfloor$ and another with $x_k \geq \lceil u \rceil$. For the moment, think of these as left and right children of p. In principle, we can create a search tree in this way and enumerate all solutions and select the optimal one, but that would be impractical. Conceptually, we will create a tree but try to avoid enumerating all solutions. In the search tree, a child has all constraints of its parents plus the additional constraint from x_k from its parent (i.e., either $x_k \leq \lfloor u \rfloor$ or $x_k \geq \lceil u \rceil$). Each node in the search tree therefore has limits on each variable. Initially we have $0 \leq x_i \leq \infty$ for $0 \leq i < n$. These limits are then added to a node (and its descendants) as constraints expressed in a node's $\mathbf{A}$ and $\mathbf{b}$.

When we have created the two left and right children of a node q, q is no longer needed and all its memory can be recycled. We need a set of nodes which wait for being explored. Denote this set by h. By exploring a node q is meant that we check if q has an integer solution which is better than what we have seen so far, in which case we remember that solution but create no children of q (since no additional constraints on the children of q can possibly find a better solution than $q.z$). If q instead was not an integer solution, we compare $q.z$ with the value of the best integer solution found so far. If $q.z$ is higher, then we create two new nodes, and add them to h, since one or both of these may lead to a better integer solution. If $q.z$ is lower, there is no point in creating any new node since they cannot be better than q.

In addition, whenever we find a better integer solution, we remove from h all waiting nodes with a lower z. The approach to solve integer linear problem we have just seen can be generalized for other problems as well. It is an example of a **branch-and-bound algorithm**. In the next section we present pseudo code for a integer program solver. The function *intopt* is the main function. In the pseudo code, delete means deallocate memory. The pseudo code should be viewed as a reference implementation as it is not optimized (translating the pseudo code to C and optimizing it is studied in the course *EDAG01 Efficient C* at Lund University). The memory used by a node for $\mathbf{A}$, $\mathbf{b}$, $\mathbf{c}$, and $\mathbf{x}$ is needed only for Simplex and can be deallocated either directly after a node no longer is needed, or, for a node which will branch, after the value of the branching variable, x_h, has been

saved. See *branch*. It should be noted that integer problems often are sparse and instead of explicitly storing and computing with all zeroes, it is then better to use a more compact representation, as discussed in Section 17.5, on page 663.

Example B.3.1 Consider an undirected graph $G(V, E)$ with m edges and n vertices. At most n colors are needed and we create n binary decision variables w_i for $0 \leq i < n$, with the meaning $w_i = 0$ if color i is unused and $w_i = 1$ if at least one vertex is assigned color i. For each vertex v we then create n decision variables x_{vi}, also binary, with the meaning that $x_{vi} = 1$ if vertex v is assigned color i and otherwise zero. The object function is to minimize the number of used colors, i.e., the sum of all w_i. There are two types of constraints:

- each vertex must be assigned a color, so the sum of all x_{vi} for a particular vertex v must be one, and
- two neighbors cannot use the same color, so if u and v are neighbors $x_{ui} + x_{vi} \leq w_i$, for all i.

The model becomes:

$$\begin{aligned}
\min \quad z \quad &= w_0 + w_1 + \ldots + w_{n-1} \\
\sum_{i=0}^{n-1} x_{vi} &= 1 \qquad \forall v \in V \\
x_{ui} + x_{vi} &\leq w_i \qquad \forall (u, v) \in E \\
x_{vi}, w_i &\in \{0, 1\} \quad \forall v \in V, 0 \leq i < n
\end{aligned}$$

Since we expect the constraints to be on the form $\mathbf{Ax} \leq \mathbf{b}$, we write it as follows instead:

$$\begin{aligned}
\min \quad z \quad &= w_0 + w_1 + \ldots + w_{n-1} \\
\sum_{i=0}^{n-1} x_{vi} &\leq 1 \qquad \forall v \in V \\
\sum_{i=0}^{n-1} -x_{vi} &\leq -1 \qquad \forall v \in V \\
x_{ui} + x_{vi} - w_i &\leq 0 \qquad \forall (u, v) \in E \\
x_{vi}, w_i &\leq 1 \qquad \forall v \in V, 0 \leq i < n
\end{aligned}$$

■

Example B.3.2 Consider next the problem of **instruction scheduling** to minimize pipeline delays. See Section 3.3, on page 156, Section 3.4, on page 158, and [66]. This is an important part of all optimizing compilers. Assume for simplicity the input is a list of instructions, each with two source operands, one destination operand and a number indicating how many clock cycles it takes a CPU to execute the instruction. For instance, for a floating point add instruction, this **latency** may be five clock cycles. The input can be described as a weighted directed acyclic graph with nodes being instructions and with an edge from u to v if instruction u computes the value of an operand of instruction v. The edge has weight the latency of u. The instruction scheduling problem is NP-complete for most realistic CPUs. How can we solve it with integer linear programming? Let our goal be to find an m cycle schedule, and that our CPU can issue r instructions each clock cycle.

Let x_i^j be a decision variable which says instruction i is scheduled in cycle j. Our schedule must satisfy the dependences in the dag, i.e., with an edge from u to v, u must not be scheduled after v for correctness, but preferably sufficiently earlier than v to take the latency of u into account. Denote by L_{uv} this latency. In this example we look for any feasible solution. With the nodes V, with $n = |V|$, and edges E, a basic model then becomes:

$$\sum_{j=1}^{m} x_i^j = 1 \qquad \forall x_i \in V$$

$$\sum_{i=1}^{n} x_i^j \leq r \qquad \forall 1 \leq j \leq n$$

$$\sum_{j=1}^{m} j \times x_k^j + L_{ki} \leq \sum_{j=1}^{m} j \times x_i^j \qquad \forall (k,i) \in E$$

In the last constraint, the expression $\sum_{j=1}^{m} j \times x_k^j$ is the cycle in which instruction k is scheduled. Wilken presents improvements to this basic formulation [78]. ■

B.4 Branch-and-bound algorithm

Algorithm B.2 Branch-and-bound for integer linear programming.

Integer linear programming algorithm.

```
struct node_t {
    int       m          /* Constraints. */
    int       n          /* Decision variables. */
    int       k          /* Parent branches on x_k. */
    int       h          /* Branch on x_h. */
    double    xh         /* x_h. */
    double    ak         /* Parent a_k. */
    double    bk         /* Parent b_k. */
    double    min[n]     /* Lower bounds. */
    double    max[n]     /* Upper bounds. */
    double    a[m][n]    /* A. */
    double    b[m];      /* b. */
    double    x[n];      /* x. */
    double    c[n];      /* c. */
    double    z;         /* z. */
}


function initial_node(m, n, a, b, c)
begin
    auto p = allocate memory for a node
    p.a = new double [m+1][n+1]
    p.b = new double [m+1]
    p.c = new double [n+1]
    p.x = new double [n+1]
    p.min = new double [n]
    p.max = new double [n]
    p.m = m
    p.n = n
    copy a, b, and c parameters to p
    for (i = 0; i < n; i = i + 1)
        p.min[i] = −∞
        p.max[i] = +∞
    return p
end
```

```
function extend(p, m, n, a, b, c, k, ak, bk)
begin
    auto    q = allocate memory for a node
    int     i,j
    q.k = k
    q.ak = ak
    q.bk = bk
    if ak > 0  and p.max[k] < ∞ then
        q.m = p.m
    else if ak < 0  and p.min[k] > 0 then
        q.m = p.m
    else
        q.m = p.m + 1
    q.n = p.n
    q.h = -1
    q.a = new double [q.m+1][q.n+1] // note normally q.m > m
    q.b = new double [q.m+1]
    q.c = new double [q.n+1]
    q.x = new double [q.n+1]
    q.min = new double [n]
    q.max = new double [n]
    copy p.min and p.max to q // each element and not only pointers
    copy m first rows of parameter a to q.a // each element
    copy m first elements of parameter b to q.b
    copy parameter c to q.c // each element
    if ak > 0 then
        if q.max[k] = ∞  or bk < q.max[k] then
            q.max[k] = bk
    else if q.min[k] = −∞  or -bk > q.min[k] then
        q.min[k] = -bk
    for (i = m, j = 0; j < n; j = j + 1) {
        if q.min[j] > −∞ then
            q.a[i][j] = -1
            q.b[i] = -q.min[j]
            i += 1
        if q.max[j] < ∞ then
            q.a[i][j] = 1
            q.b[i] = q.max[j]
            i += 1
    }
    return q
end
```

```
function is_integer(xp)
begin
    // xp is a pointer to a double
    double x = *xp
    double r = round(x) // ISO C lround
    if |r − x| < ϵ then
        *xp = r
        return 1
    else
        return 0
end

function integer(p)
begin
    int i
    for (i = 0; i < p.n; i = i + 1)
        if !is_integer(&p.x[i]) then
            return 0
    return 1
end

procedure bound(p, h, zp, x)
    // zp is a pointer to max z found so far
    if p.z > *zp then
        *zp = p.z
        copy each element of p.x to x // save best x
        remove and delete all nodes q in h with q.z < p.z
end

function isfinite(x)
begin
    // ISO C function
    if x is a NaN  or |x| = ∞ then
        return 0
    else
        return 1
end
```

```
function branch(q,z)
begin
    double min,max
    if q.z < z then
        return 0
    for (h = 0; h < q.n; h = h + 1)
        if !is_integer(&q.x[h]) then
            if q.min[h] = −∞ then
                min = 0
            else
                min = q.min[h]
            max = q.max[h]
            if ⌊q.x[h]⌋ < min  or ⌈q.x[h]⌉ > max then
                continue
            q.h = h
            q.xh = q.x[h]
            delete each of a,b,c,x of q // or recycle in other way
            return 1
    return 0
end



procedure succ(p,h,m,n,a,b,c,k,ak,bk,zp,x)
    auto q = extend(p,m,n,a,b,c,k,ak,bk)
    if q = null then
        return
    q.z = simplex(q.m, q.n, q.a, q.b, q.c, q.x, 0)
    if isfinite(q.z) then
        if integer(q) then
            bound(q,h,zp,x)
        else if branch(q, *zp) then
            add q to h
            return
    delete q
end
```

```
function intopt(m, n, a, b, c, x)
begin
    auto p = initial_node(m,n,a,b,c)
    set h = {p}
    double z = −∞ // best integer solution found so far
    p.z = simplex(p.m, p.n, p.a, p.b, p.c, p.x, 0)
    if integer(p)  or !isfinite(p.z) then
        z = p.z
        if integer(p) then
            copy p.x to x
        delete p
        return z
    branch(p,z)
    while h ≠ ∅
        take p from h
        succ(p, h, m, n, a, b, c, p.h, 1, ⌊p.xh⌋, &z, x)
        succ(p, h, m, n, a, b, c, p.h, -1, −⌈p.xh⌉, &z, x)
        delete p
    if z = −∞ then
        return NaN // not-a-number
    else
        return z
end
```