## 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 [m+n+1]
    p.min = new double [n]
    p.max = new double [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
    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+1]
    q.max = new double [n+1]
    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
    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
    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,x)
    set h = {p}
    double z = −∞ // best integer solution found so far
    p.z = simplex(p.m, p.n, p.a, p.b, p.c, q.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, ⌊x.h⌋, &z, x)
        succ(p, h, m, n, a, b, c, p.h, -1, −⌈x.h⌉, &z, x)
        delete p
    if z = −∞ then
        return NaN // not-a-number
    else
        return z
end
```