The Branch and Bound Algorithm

The Branch and Bound (B&B) algorithm is a general method to solve discrete optimization problems. The key idea is that B&B attempts to prune parts of the exhaustive search tree by using bounds on the value of the optimum solution. Below we present a formalized description of the method. In order to understand how it works, let us review first how we can formalize the exhaustive search.

Consider the following optimization problem. Let B be the set of all n-dimensional binary vectors and we would like to find the maximum of a function $f(\mathbf{x})$ over B, that is, we are looking for

$$\max_{\mathbf{x}\in B} f(\mathbf{x}).$$

Note: boldface letters will be used to denote vectors, such as

$$\mathbf{x} = (x_1, \ldots, x_n), \qquad \mathbf{b} = (b_1, \ldots, b_n).$$

We can formalize the exhaustive search as a recursive procedure. To present it, let us introduce a few notations. Let $B_k(\mathbf{b})$ denote the subset of B which is obtained by fixing the first k coordinates of the binary vectors according to the coordinates of a given binary vector $\mathbf{b}$:

$$B_k(\mathbf{b}) = \{\mathbf{x} \in B \,|\, x_1 = b_1, \ldots, x_k = b_k\}.$$

That is, the first coordinate is fixed at b_1, the second at b_2, ... the k^{th} is fixed at b_k and the rest of the coordinates of $\mathbf{x}$ is arbitrary (0 or 1). If $k = 0$, then it means we did not fix anything, so $B_0(\mathbf{b}) = B$. If $k = n$, then all coordinates are fixed at the corresponding b_i value, so then the set $B_n(\mathbf{b})$ contains only a single vector, $\mathbf{b}$ itself: $B_n(\mathbf{b}) = \{\mathbf{b}\}$.

Let us introduce now the following family of functions:

$$F_k(\mathbf{b}) = \max_{\mathbf{x}\in B_k(\mathbf{b})} f(\mathbf{x}).$$

The meaning of $F_k(\mathbf{b})$ is that this is the maximum value of $f(\mathbf{x})$ if the maximization is done over the restricted set $B_k(\mathbf{b})$, that is, the first k coordinates of $\mathbf{x}$ are fixed at the values determined by $\mathbf{b}$ and only the rest can change.

What is the advantage of the $F_k(\mathbf{b})$ functions? The reason for we have introduced them is that we can easily construct a recursive algorithm to compute them. Once we have this subroutine that computes $F_k(\mathbf{b})$ for any k and $\mathbf{b}$, then we can solve the original problem by simply calling the subroutine with $k = 0$, since, by definition we have

$$F_0(\mathbf{b}) = \max_{\mathbf{x} \in B_0(\mathbf{b})} f(\mathbf{x}) = \max_{\mathbf{x} \in B} f(\mathbf{x})$$

for any $\mathbf{b}$ (so we can use, for example, $\mathbf{b} = \mathbf{0}$ in this call).

Let us see now how we can compute the $F_k(\mathbf{b})$ values recursively. We present an algorithm in pseudo-code below first for the *exhaustive search*.

Exhaustive Search

```
1.  function F_k(b)
2.     if k = n then return f(b)
3.     else
4.       begin
5.          b_{k+1} := 0;   u := F_{k+1}(b)
6.          b_{k+1} := 1;   v := F_{k+1}(b)
7.          return max(u, v)
8.        end
```

Explanation: In line 2 we handle the case when all coordinates are fixed, i.e., we have arrived at a leaf of the search tree. In this case $B_n(\mathbf{b}) = \{\mathbf{b}\}$, so the maximum over this set can only be $f(\mathbf{b})$. If $k < n$ then in lines 5 and 6 we compute recursively the function for two possible cases (*branch*): when the next coordinate is fixed at 0 (line 5) and when it is fixed at 1 (line 6). The maximum of the two gives the value of $F_k(\mathbf{b})$ in line 7, since this is the maximum over the set when the $(k+1)^{th}$ coordinate is not fixed, only the first k. What guarantees that the recursion ends after a finite number of steps? This is guaranteed by the fact that the recursive call is always made for a larger value of k and when the largest value $k = n$ is reached then there is no more recursive call.

Let us now turn to the Branch and Bound algorithm. Suppose that we have an *upper bound function* $\mathrm{U}_k(\mathbf{b})$ available, such that

$$F_k(\mathbf{b}) \leq \mathrm{U}_k(\mathbf{b})$$

holds for any k and $\mathbf{b}$. It is assumed that an independent subroutine is available that can compute $\mathrm{U}_k(\mathbf{b})$ for any parameter value.

Let us introduce the variable *maxlow* that will carry the largest lower bound on the global optimum that has been achieved so far. We do not need a separate subroutine for computing a lower bound, because whenever we compute a value $f(\mathbf{x})$ for a particular $\mathbf{x}$, it is automatically a lower bound on the maximum. This is so, simply because the function at any $\mathbf{x}$ is either maximum or less, that is,

$$f(\mathbf{x}) \leq \max_{\mathbf{y} \in B} f(\mathbf{y})$$

holds for any $\mathbf{x}$.

Now we can present the B&B algorithm by modifying the exhaustive search in the following manner. For simplicity, let us assume that the function $f(\mathbf{x})$ is nonnegative. Then the variable *maxlow* can be initially set to 0.

Branch and Bound

```
1.   function F_k(b)
2.      if k = n then
3.        begin
4.          if f(b) > maxlow then maxlow := f(b)
5.          return f(b)
6.        end
7.      else
8.        if U_k(b) > maxlow then
9.           begin
10.             b_{k+1} := 0;   u := F_{k+1}(b)
11.             b_{k+1} := 1;   v := F_{k+1}(b)
12.              return max(u, v)
13.           end
14.       else return maxlow
```

Explanation. Observe that if we remove lines 4, 8 and 14 then we exactly

get back the exhaustive search (ignoring the unnecessary `begin-end` pairs). Thus, let us take a closer look how these lines change the exhaustive search.

In the case $k = n$ the only change is that before returning $f(\mathbf{b})$ we update the value of *maxlow* in line 4: if the new function value is larger then the old value of *maxlow*, then this larger value will be the new best lower bound on the optimum, otherwise the old one remains.

The most essential difference is in line 8, this is the *bounding*. Here the idea is that we only make the recursive calls if the upper bound $\mathrm{U}_k(\mathbf{b})$ is larger than the current value of *maxlow*. Why we can do this without losing anything? Because if the condition in line 8 is *not* satisfied, then

$$maxlow \geq \mathrm{U}_k(\mathbf{b}) \geq F_k(\mathbf{b})$$

must hold. This means that from this parameter combination (k, $\mathbf{b}$) we cannot hope improvement on the best value found so far, since the maximum in this subtree of the search tree (i.e., $F_k(\mathbf{b})$) is already known to be not more than *maxlow*, so it makes no sense to explore this subtree. Thus, in this case we do not make any recursive call and return *maxlow* in line 14 (just to return something).

Note that for the above bounding step we need to compute the value of $\mathrm{U}_k(\mathbf{b})$, but we assumed that for this an independent subroutine is available. The savings may be essential if this subroutine is much faster then the exhaustive recursive algorithm. This is typically the case in the successful applications of B&B.

Comments:

- If we drop the assumption $f(\mathbf{x}) \geq 0$, the algorithm still remains the same, just *maxlow* should be initialized to a value for which $maxlow \leq f(\mathbf{x})$ holds for all $\mathbf{x}$.
- If the original maximization problem allows only certain binary vectors $\mathbf{x}$, then we first extend the function to all binary vectors by assigning a small enough function value to those vectors which are excluded in the original maximization problem. ("Small enough" means here a value that is surely smaller than any function value on those vectors that are not excluded in the original problem).
- If we also want to find the maximizing vector $\mathbf{x}$, not just the maximum value $f(\mathbf{x})$ (i.e., we ask not only that *how much* is the maximum, but also that *where* it occurs), then we have to keep record the latest $\mathbf{b}$ vector along with updating *maxlow*, whenever a higher value (an update) occurs in line 4.
- The algorithm is presented here for a maximization problem. If the problem is minimization, then everything should be changed accordingly, that is, then we need a lower bound function instead of an upper bound function, we maintain a minimum upper bound *minup* instead of a maximum lower bound *maxlow* etc. If you understand the algorithm, it should not be a problem to convert it into minimization.