6
Branch and Bound

A branch-and-bound algorithm was developed to solve UBQP exactly.

A branch-and-bound algorithm is an algorithm for solving $\mathcal{NP}$-Hard problems exactly. The basic build blocks of the algorithm for solving problem P are:

- Let P and P' be the problems of finding $\min(f(x)|x \in X)$ and $\min(f'(x)|x \in X')$ respectively, where X and X' are the feasible regions for P and P'.
  If $X \subseteq X'$ and $f(x) = f'(x)$ for all $x \in X$, then P' is called a relaxation of P. A relaxation is only really useful if it is easier to solve than the original problem.
  We need a relaxation P' of P.
- Given the problem P of finding $\min(f(x)|x \in X)$, the family of sets $X_1, \ldots, X_t$ is called a "branching" of P if $\cup_{i=0}^{t} X_i = X$.
  A branching of P is only useful if each problem $\min(f(x)|x \in X_i)$ is another instance of the same problem as P, so that we can solve each of these problems recursively.
  Ideally, the branching will form a partition of X.
  We need a branching of P.
- An heuristic solution to P, i.e., an algorithm that finds a "good" solution to P.
  An heuristics to P is optional. It can greatly improve the performance of the branch-and-bound.

With this in mind, we still need to proof some results, before showing the branch-and-bound.

Theorem 14 *The value of the relaxation is a lower bound to the value of the original problem.*

Proof. Let P be a problem and P' be a relaxation of it. Let x'^* be the optimal solution to P' and x^* be the optimal solution of P.
What the statement of the theorem says is that $f'(x'^*) \leq f(x^*)$. To proof it, assume this is not the case.
Well, by the definition of relaxation, $f'(x^*) = f(x^*) < f'(x'^*)$. This is a contradiction, since x'^* cannot be the optimal solution to P'.
∎

Theorem 15 *Let P and P' be a problem and its relaxation. Let x'^* be the optimal solution to P'. If $x'^* \in X$ then x'^* is also the optimal solution to P.*

Proof. Assume this is not the case, so there is $x \in X$ such that $f(x) < f(x'^*)$. In this case, from the definition of relaxation: $f'(x) = f(x) < f(x'^*) = f'(x'^*)$ which is contradictory, because x'^* cannot be the optimal solution to P'.
∎

Theorem 16 *Let $X_1, \ldots, X_t$ be a branching of the problem P. Then the objective value of the problem P is the minimum of the objective values of the problems* $\min(f(x)|x \in X_i)$.

Proof. Since $\cup_{i=1}^{t} X_i = X$, the optimal solution x^* of P must be in at least one of X_i. ∎

Finally, with these results in mind it is time to show the general branch-and-bound algorithm for the problem P. The *incumbent* is the value of the best known solution. It can be initialized with any $x \in X$

Data: X, h (the height in the branch-and-bound tree)
Result: $x^* \in X$ such that $\forall x \in X f(x^*) \leq f(x)$
$xlb \leftarrow$ solve-relaxation(X);
`//` xlb `is a lower bound to` P `because of 14`
if $f'(xlb) \geq f(incumbent)$ **then**
  return *incumbent*
end
if $xlb \in X$ **then**
  $incumbent \leftarrow xlb$ **return** xlb
end
$xub \leftarrow$ heuristics(X);
`// this step is optional`
if $f(xub) < f(incumbent)$ **then**
  $incumbent \leftarrow xub$
end
$X_1, \ldots, X_t \leftarrow$ branching(X); `// Do the actual branching`
forall the $i = 1, \ldots, t$ **do**
  call_recursively(X_i, $h + 1$)
end
return (*incumbent*)

Algorithm 1: Branch and Bound

Now, in the next few sections I will explain how the branch-and-bound algorithm works for the UBQP using the method described in chapter 5.

6.1 Relaxation

Our relaxation of P was made in several steps:

1. We found the lower bound lb_1 achieved by 3.3.
2. For each x_i we calculate $K_0(i)$ (and $K_1(i)$) which means "the lower bound if we fix x_i to 1 (to 0)". There are further explanation on 6.1.1.
3. The actual best lower bound is $\max(lb_1, \max_i(\min(K_0(i), K_1(i))))$.

6.1.1 Computing $K_0(i)$ and $K_1(i)$

There are several ways of computing these lower bounds.

One way is to fix x_i (to 0 or to 1) and then find the lower bound using 3.3. We will call this the "slow" method.

Another way is computing $U_0(i)$ and $U_1(i)$ as stated in 5.2 for λ in a family Λ. This will be called "fast" method.

The Λ used consists of $1000e_i$ for each i and the λ given by 3.3 on the original problem (without any variable fixing).

One can easily see that the slow method yields to better lower bounds than the fast method, because while the former fixes $x_i \geq 0$ ($x_i \leq 1$), tha later fixes $x_i = 1$ ($x_i = 0$).

But the fast method, as the name suggests, is much faster than the slow method.

Experiments showed that the best is to randomly use one or the other. The probability of running the slow method is $\frac{1}{1.3^h}$ where h is the height of the node in the branch-and-bound tree (the root has $h = 0$). It is intuitive that it is better to spend a good time in the upper part of the branch-and-bound tree, and this probability makes exactly this.

Besides that, with probability $0.7^{\frac{h}{2}}$ we further improve the bound by constraining $0 \leq x_i \leq 1$. This can be done in both the slow and the fast

method, by using the Lagrangian of the new problem.

Of course, if for some variable $K_0(i) \geq f(incumbent)$ or $K_1(i) \geq f(incumbent)$ we can fix this variable to 1 or to 0.

6.2 Branching

The branching used was the simplest of them, we partition X as X_1, X_2, where in X_1 a variable x_i is set to 0 and in X_2 the same variable x_i is set to 1. The "hard" part was the choice of the variable x_i on which we would branch on.

For this purpose we used the bounds $K_0(i)$ and $K_1(i)$ computed at 6.1. They can be used in several intuitive ways, such as:

- Branch on the variable x_i that has the maximum $\max(K_0(i), K_1(i))$, and solve first the branch that corresponds to the maximum of these two.
  The intuition behind this decision is that this choice is the closest to the best known upper bound, so it will be pruned very quickly, or, if we get lucky, solved to optimality very quickly.

- Branch on the variable x_i that has the minimum $\min(K_0(i), K_1(i))$, and solve first the branch that corresponds to the minimum of these two.
  This way we are branching to the least lower bound, so one can expect that the best solution for this branch will be very good, which will give a good incumbent solution. The best the incumbent, the more pruning is done.

- Branch on the variable x_i that has the maximum absolute difference between $K_0(i)$ and $K_1(i)$.
  This way we are branching on the variable that is "more certain", that is, one can expect that $x_i = 0$ if $K_1(i) < K_0(i)$ and that $x_i = 1$ if $K_0(i) < K_1(i)$. So we solve first the branch that corresponds to the minimum of these two.

Experimentally the last way gave the best results.

6.3 Heuristics

The heuristics used was a simple VDS (Variable Depth Search) that acts as follows:

Data: X, which can be interpreted as the set of variables
Result: $x \in \{0,1\}^X$ which has a "low" $f(x)$
$times \leftarrow 0$ $best \leftarrow 0^X$ **while** $times < T$ **do**
  $\sigma \leftarrow$ a random permutation of X;
  $tempbest \leftarrow best$;
  $temp \leftarrow best$;
  forall the $i \in \sigma$ *in the order of* σ **do**
    Swap the value of $temp_i$;
    if $f(temp) < f(tempbest)$ **then**
      $tempbest \leftarrow temp$;
    end
  end
  if $f(tempbest) < f(best)$ **then**
    $best \leftarrow tempbest$;
  end
end
return $best$;

Algorithm 2: Heuristics

This heuristics was enhanced by CGI (explained in chapter 4).

Also, if the number of variables is less than 25, we find the optimal solution exhaustively, using a brute-force algorithm.

6.4
A slighty modification

We also made a greedy heuristics by slightly modifying the branch and bound. The modifications were:

- We don't run the heuristics when solving the nodes.

- We branch using the "second way" described in section 6.2. Namely, we branch on the variable with the minimum $\min(K_0(i), K_1(i))$, and solve first the branch corresponding to the minimum of these two.

- Whenever we reach a feasible solution (normally after running the brute-force) we stop the algorithm and return it as the solution.

This is a simple greedy heuristics, but leads to some interesting results, stated on section 7.2.