# 20191107

# Branch and Bound Continued

Let's revisit the B&B algorithm:

- Characterize the solution as a series of decisions. This can be as simple as iterating through the set of items, deciding to include or omit each one in turn. We should specify the order in which the decisions will be made.
- Establish a Global Upper Bound on the cost of the optimal solution. Call this $U_{Global}$
- Create a set $\mathcal{P}$ of feasible partial solutions representing the possible outcomes of the first decision. For each $P \in \mathcal{P}$, determine the cost bracket $[l_P, u_P]$
- Execute the following:

```
while P is not empty:
    Choose P ∈ P such that l_P is the smallest in P
    if P is a complete solution:
        return P      # P is an optimal solution
    else:
        delete P from P
        for each new feasible partial or full solution Q that can be constructed from
        P as a result of the next decision to be made:
            compute [l_Q, u_Q]
            if  l_Q > U_Global:
                discard Q
            else:
                if  u_Q < U_Global:
                    U_Global = u_Q
                add Q to P
```

By this point the rationale for discarding partial solutions with $u_Q > U_{Global}$ should be clear. But we still need to think about how we can be sure that that solution we return is the optimal solution. It is certainly true that there may still be a lot of unexplored partial

solutions in $\mathcal{P}$ when we return. How can we be sure that none of them can lead to a solution that is better than the one we choose?

The solution $P$ that we return has two characteristics:

- its lower bound $l_P$ is $\leq$ all other lower bounds in $\mathcal{P}$
- it is a complete solution

When we say $P$ is a complete solution, we mean that $P$ consists of a complete sequence of decisions – there is nothing more to be decided. This means that we know exactly the cost of $P$. In terms of our notation, it means $l_P = u_P$

But this means $u_P \leq l_Q \quad \forall \quad Q \in \mathcal{P}$

Which means that every extension of every partial solution still in $\mathcal{P}$ will cost at least as much as $P$ costs. Therefore no solution can be better than $P$.

Now we need to think about the implementation of the algorithm. The computation of the lower and upper bounds depends on the particular problem being solved. Our goal is get good bounds (ie bounds that are close together) without doing too much work. It's not uncommon to do O($n^2$) work when computing bounds for a partial solution.

The one part of the algorithm we haven't looked at carefully is :

Choose $P \in \mathcal{P}$ such that $l_P$ is the smallest in $\mathcal{P}$

It's deceptively simple-looking. But we need to remember that we typically apply B&B to problems where the number of potential solutions is exponential. This suggests that we may have O($2^n$) partial solutions in $\mathcal{P}$. (Obviously we hope that our bounding operations allow us to keep $\mathcal{P}$ small, but we can't be sure that they will.) So how do we store $\mathcal{P}$ ?

Whatever structure we choose has to support two operations: we need to able to add new items to the set, and we need to be able to select and remove the item with the smallest $l_P$. You might be thinking that we also need to delete partial solutions when they become obsolete (ie when $U_{Global}$ drops below a partial solution Q's $l_Q$ value). We'll come back to that.

One option is to store $\mathcal{P}$ in a list or array. When we generate new partial solutions, we can just add them to the end – that takes constant time. But then finding P with the smallest $l_P$

requires looking at everything in $\mathcal{P}$ ... which may be O($2^n$) in size. Many programming languages have built-in functions with names like "smallest" that will return the lowest value in a set, but this is what they are doing – just looking at the elements of the set one by one. If we take this approach we are embedding a potentially O($2^n$) operation in every iteration of the while loop.

What if we sort the array (or list) containing $\mathcal{P}$ ? That would make the task of finding the P we want very simple – it's the first one – we can find it in O(1) time. But now we have to consider the problem of adding new partial solutions to $\mathcal{P}$ . If they need to be inserted in the middle of the sorted list (or array) that requires O($2^n$) operations. If we add them at the end and then just re-sort the whole set, that takes O($n * 2^n$) operations.

It's not looking good. If only there were some data structure that lets us find the smallest value quickly, ***and*** add new values quickly.

The solution of course is to use a min-heap. A min-heap keeps the smallest value at the top so finding it is an O(1) operation. When we remove the value from the top, fixing the heap (which some people seriously call "re-heapifying") takes O($\log t$) steps, where t is the number of items in the heap. Similarly, adding new items to the heap takes O($\log t$) steps where t is the number of items in the heap.

And $\log 2^n = n$ !!! (Not n-triple-factorial, just n-very-excited) So now, even if $\mathcal{P}$ grows to have O($2^n$) partial solutions, we can do all of our necessary operations on it in O(n) time.

This is my favourite illustration of how choosing the right data-structure can have a massive effect on the efficiency of an algorithm.

Now what about the idea of removing partial solutions from $\mathcal{P}$ when we know they cannot lead to an optimal solution? We can do that – we can traverse the heap either from the top down or the bottom up, removing any item that we don't need, and all of its descendants too. The problem is that this can damage the heap structure – we would need to rebuild the heap from scratch, which takes O(t) where t is the number of elements in the set ... which can be O($2^n$) as we know. But if we just leave these obsolete partial solutions in the heap they don't really cause much trouble – remember we can do the operations we need in O(n) time even if the heap is really big. On balance it seems better to leave the partial solutions with high $l_P$ values in the heap, even though they are no longer of any use to us. The only exception is when memory space is limited. In this situation it may be necessary to do some clean-up on the heap from time to time.