Branch and Bound Algorithms

(This material is not covered in the text. See the "Recommended Readings" for some online resources.)

The year is 1927.

You are excavating in the Valley of the Kings, in Egypt.

You have found the start of a path to King Tut's tomb, but your map is incomplete.

Branch and Bound Algorithms

Your map guarantees that there is a route that will get you to Tut's Tomb in no more than 9 hours.

You send out a scout, and after travelling for **1** hour she comes to a crossroad.

She calls to say there are three ways she could go.

How to choose?

She tells you there are sign posts on the paths:

The sign on the first path says

Tut's Tomb : between 2 and 17 hours from here

The sign on the second path says

Tut's Tomb: between 4 and 10 hours from here

The sign on the third path says

Tut's Tomb: between 10 and 12 hours from here

Let’s take a look at that map ...

This illustrates the essential characteristics of a **branch and bound** solution.

1. The problem to be solved is an optimisation problem in which we have to make a sequence of decisions. WLOG, assume we are trying to **minimise** the objective function.

2. There is an initial upper bound on the optimal solution.

3. For any feasible partial solution P, we can compute two things:
   - a lower bound on the cost of the **best** solution that can be built from P
   - an upper bound on the cost of the **best** solution that can be built from P

We can think of the algorithm as a form of intelligent back-tracking.

We keep track of partial solutions (usually conceptualising them as a tree).

For each partial solution, we compute bounds on the complete solutions obtainable from that point.

At each step, we choose a partial solution to expand.

We eliminate partial solutions that cannot lead to the optimal solution.

We update our information about the optimal solution.

We can think of the algorithm as a form of intelligent back-tracking.

We keep track of partial solutions (usually conceptualising them as a tree).

For each partial solution, we compute bounds on the complete solutions obtainable from that point.

At each step, we choose a partial solution to expand.

We eliminate partial solutions that cannot lead to the optimal solution.

We update our information about the optimal solution.

We can think of the algorithm as a form of intelligent back-tracking.

We keep track of partial solutions (usually conceptualising them as a tree).

For each partial solution, we compute bounds on the complete solutions obtainable from that point.

At each step, we choose a partial solution to expand.

We eliminate partial solutions that cannot lead to the optimal solution.

We update our information about the optimal solution.

We can think of the algorithm as a form of intelligent back-tracking.

We keep track of partial solutions (usually conceptualising them as a tree).

For each partial solution, we compute bounds on the complete solutions obtainable from that point.

At each step, we choose a partial solution to expand.

We eliminate partial solutions that cannot lead to the optimal solution.

We update our information about the optimal solution.

We can think of the algorithm as a form of intelligent back-tracking.

We keep track of partial solutions (usually conceptualising them as a tree).

For each partial solution, we compute bounds on the complete solutions obtainable from that point.

At each step, we choose a partial solution to expand.

We eliminate partial solutions that cannot lead to the optimal solution.

We update our information about the optimal solution.

We can think of the algorithm as a form of intelligent back-tracking.

We keep track of partial solutions (usually conceptualising them as a tree).

For each partial solution, we compute bounds on the best complete solution obtainable from that point.

At each step, we choose a partial solution to expand.

We eliminate partial solutions that cannot lead to the optimal solution.

We update our information about the optimal solution.

Let **U** be an upper bound on the cost of the optimal solution. **U** can be obtained by randomly generating an arbitrary solution to the problem, and using its cost as **U**.

Let S be the set of partial solutions still under consideration.

Initially S can consist of all possible "first choices", or S can contain just one element: the partial solution in which no choice has been made.

For each P in S, let (L_P,U_P) be the bounds on the best possible solution that can be built from P.

While S is non-empty

Choose some P in S. (*different choice rules can be used)*

S = S \ {P} *(remove P from S)*

For Each P' that can be built from P with one more decision,

compute ($L_{P'}$,$U_{P'}$)

If $L_{P'}$ > **U**, discard P'

Else

If P' is a partial solution, S = S + {P'}

Elsif P' is a full solution with a better cost than the best full solution seen so far, remember P' as the best full solution

If $U_{P'}$ < **U**, **U** = $U_{P'}$ *(update global **U**)*

End For Each

End While

Return the solution being remembered

Practical Considerations

Choosing the partial solution to expand:

Depth first - choose the best child of the most recently expanded partial solution, if any
- if none, back up to the parent and try from there

Breadth first - choose a partial solution closest to the root of the solution tree

Best first - choose the partial solution with the lowest lower bound

Best first is the most used – but we need to think about how to manage the set of "live" partial solutions so that we can quickly choose the one with the lowest lower bound.

Hmmm, what data structure is **really good** for giving quick access to the smallest value in a set?

One method is to store the partial solutions in a **min-heap:**

Each new item can be inserted in O(log t) time, where t is the number of partial solutions in the heap.

Each choice for the next partial solution to be expanded can be extracted in O(log t) time.

Since t may be O(2^n) where n is the number of decisions to be made, this gives us O(n) time for selecting the next partial solution and for inserting new partial solutions.

Lower and Upper Bounds

The more accurate the lower and upper bounds for each partial solution are, the more effectively the bad branches can be pruned.

In some applications it may be worth using quite complex algorithms to compute good bounds.

For each partial solution P, the lower bound consists of two parts:

Cost so far: the cost of decisions already made

Guaranteed future cost: unavoidable costs from future decisions

The upper bound also consists of two parts:

Cost so far: same as above

Feasible future cost: the cost of some extension of P to a complete solution

The quality of the initial upper bound can be critically important.

Rather than randomly choosing a solution to give the initial upper bound, it is sometimes worthwhile to invest the time to find a fairly good solution for this purpose.

This can be done with an heuristic algorithm that runs in polynomial time but doesn't always find the optimal answer.

For example, we might be solving a problem for which there is no greedy algorithm solution. However, we might use a greedy algorithm to get the initial upper bound, and then use branch and bound to find the optimal solution.

Why Use "Best First"?

Recall that "best first" means "choose the element of S with the lowest lower bound".

The best expansion for any given partial solution may have a total cost **anywhere** between its bounds.

So the "best first" strategy may not lead directly to the optimal solution … but it has a huge advantage: we can stop before S is empty!

Generate a solution to get the initial **U** value; remember this as best_solution
Initialize S
while S is non-empty:
    Choose P in S with minimum L_P
    if $\mathbf{L_P} > \mathbf{U}$ **:**
        Break **# Optimal Solution has been found**
    else:
        S = S \ {P}
        for each P' that can be built from P with one more decision,
            Compute ($L_{P'}$,$U_{P'}$)
            if P' is a full solution:
                compare it to best_solution and update best_solution if needed
            else if $L_{P'} > \mathbf{U}$**:**
                discard P'
            else
                S = S + {P'}
            if $U_{P'} < \mathbf{U}$, $\mathbf{U} = U_{P'}$
        end for each
end while
Return best_solution

The big differences between this version of the algorithm and the original are:

- now we return full solutions back into S, instead of keeping track of the best one we have seen
- As soon as a full solution in S is selected, we stop (even if S is still full of partial solutions)

This is valid because when a full solution P is generated, its lower bound and upper bound are equal (there is no more uncertainty), and when a full solution is **selected**, its cost is <= the best possible expansion of all other items in S (this is how P is chosen). Thus there can be no other solution that beats this one.

Let's do an example!

The 0-1 Knapsack Problem: We have a collection of objects, each with a known volume and a known value. We have a knapsack with a known capacity. We want to choose the most valuable set of objects that will fit in the knapsack.

We know this is an NP-Complete problem.

With n objects to choose from, there are potentially 2^n possible solutions to be considered (every subset of the set of objects).

But with a branch and bound algorithm, we can try to cut this down a bit.

First, what is our objective function?

The obvious one is to compute the value of items chosen, and try to maximise it ...

First, what is our objective function?

The obvious one is to compute the value of items chosen, and try to maximise it …

… except that we have developed the algorithm in terms of minimisation.

First, what is our objective function?

The obvious one is to compute the value of objects chosen, and try to maximise it …

… except that we have developed the algorithm in terms of minimisation.

So let's compute the value of the objects *not* chosen - minimising this will maximise the value of the set of objects we choose.

Second, how can we conceptualise this as a sequence of decisions?

Second, how can we conceptualise this as a sequence of decisions?

Easy - list the objects in some order. At each stage, we make the decision to include the next object or not.

Choosing a solution to get an initial upper bound:

We can use a simple greedy algorithm, based on choosing the object with the maximum ratio of value to volume.

Computing lower and upper bounds for partial solutions:

Each partial solution contains a "cost so far" - the value of all items already excluded. This certainly works as a lower bound on the cost of all extensions of the partial solution … but we can do better. How?

Computing lower bounds for partial solutions:

Each partial solution contains a "cost so far" - the value of all items already excluded. This certainly works as a lower bound on the cost of all extensions of the partial solution … but we can do better. How?

We can exclude all objects yet to be considered which will not fit in the knapsack on top of the objects already chosen.

Computing upper bounds for partial solutions:

The “cost so far” obviously contributes to the upper bound, and a simple and valid extension is to imagine that we will also leave out of the knapsack all the objects not yet considered. But we can do better than that ...

Computing upper bounds for partial solutions:

The "cost so far" obviously contributes to the upper bound, and a simple and valid extension is to imagine that we will also leave out of the knapsack all the objects not yet considered. But we can do better than that ...

Remember that **any solution** gives us an upper bound on the cost of an **optimal solution** ...

Computing upper bounds for partial solutions:

The "cost so far" obviously contributes to the upper bound, and a simple and valid extension is to imagine that we will also leave out of the knapsack all the objects not yet considered. But we can do better than that ...

Remember that **any solution** gives us an upper bound on the cost of an **optimal solution** …

so applying the greedy heuristic to the remaining objects will give us a better upper bound for the current partial solution.

Please see the posted spreadsheet for a fully worked out example of developing a B&B algorithm for the 01-Knapsack Problem.