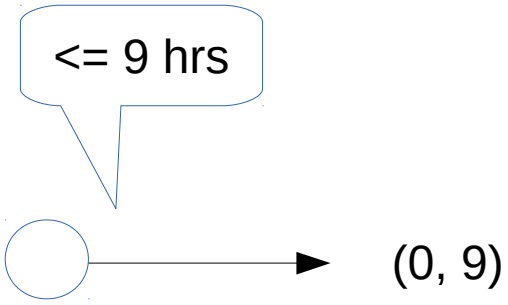
20191101

Branch and Bound

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

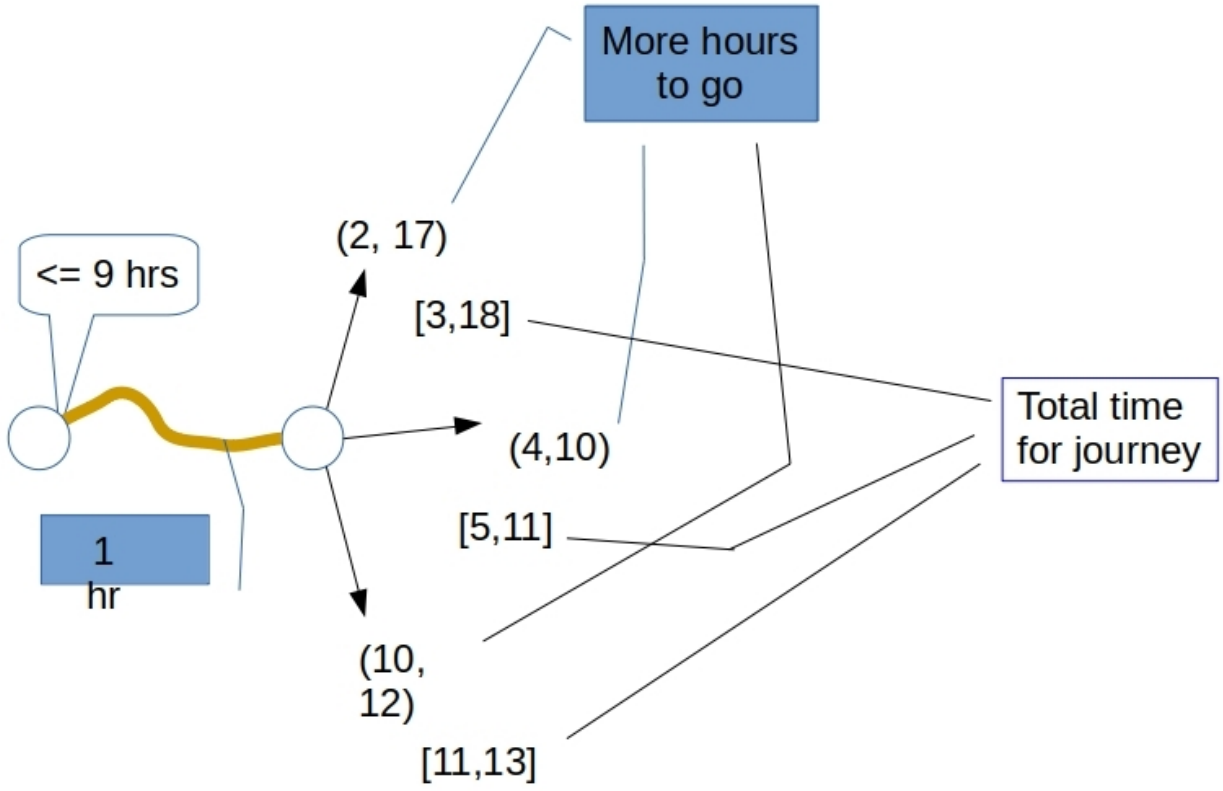
The year is 1927.

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.



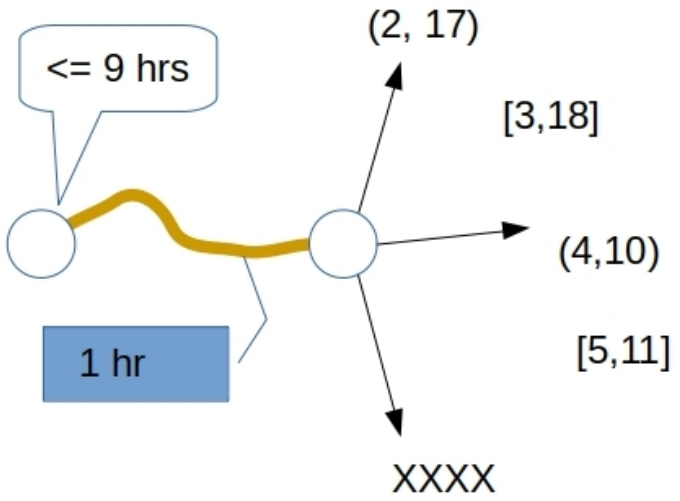
It shows the start of the path, and states that the shortest route takes ≤ 9 hours. We use the notation (0,9) to indicate that the trip will take between 0 and 9 hours.

Rather than explore in person, you send out a scout. After an hour you receive a message back from the scout – they have reached a cross-road and don't know which way to go. There are signs on all three possible paths.

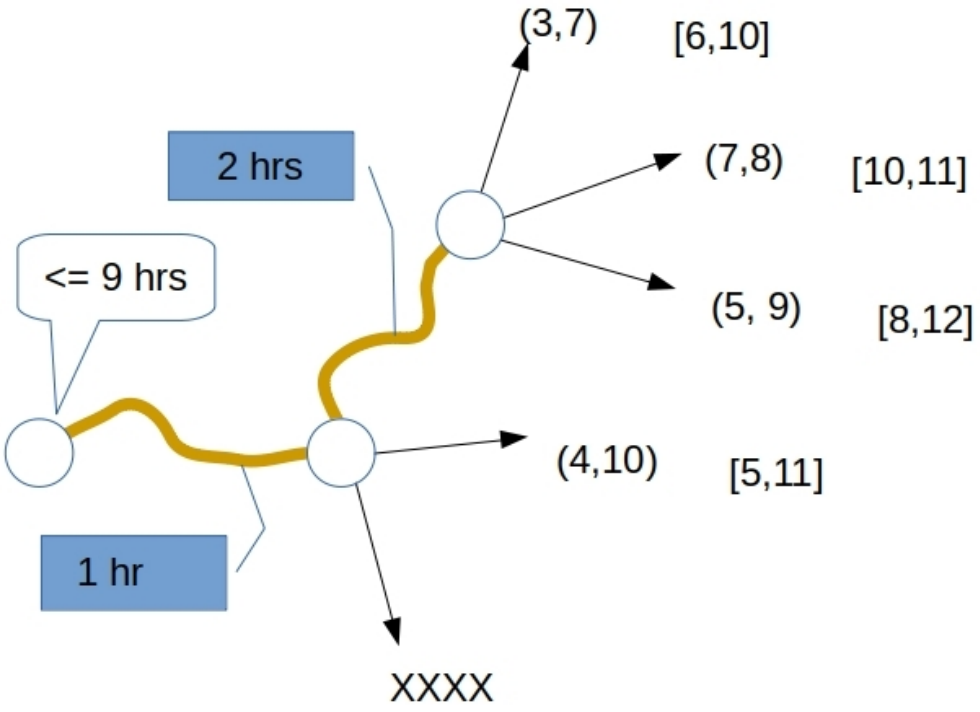


One sign says (2, 17), which means following that path will get you to Tut's tomb in somewhere between 2 hours and 17 hours. Another says (4, 10) which means somewhere between 4 and 10 more hours. The third says (10, 12) which means somewhere between 10 and 12 hours. Since it took 1 hour to reach this cross-road we can label the three paths with the total-journey time bounds shown in square brackets.

The path labelled [11, 13] can be ignored. Its best possibility is a trip of 11 hours and we already know we can get to Tut's tomb in ≤ 9 hours ... so the best route can't be along this path.

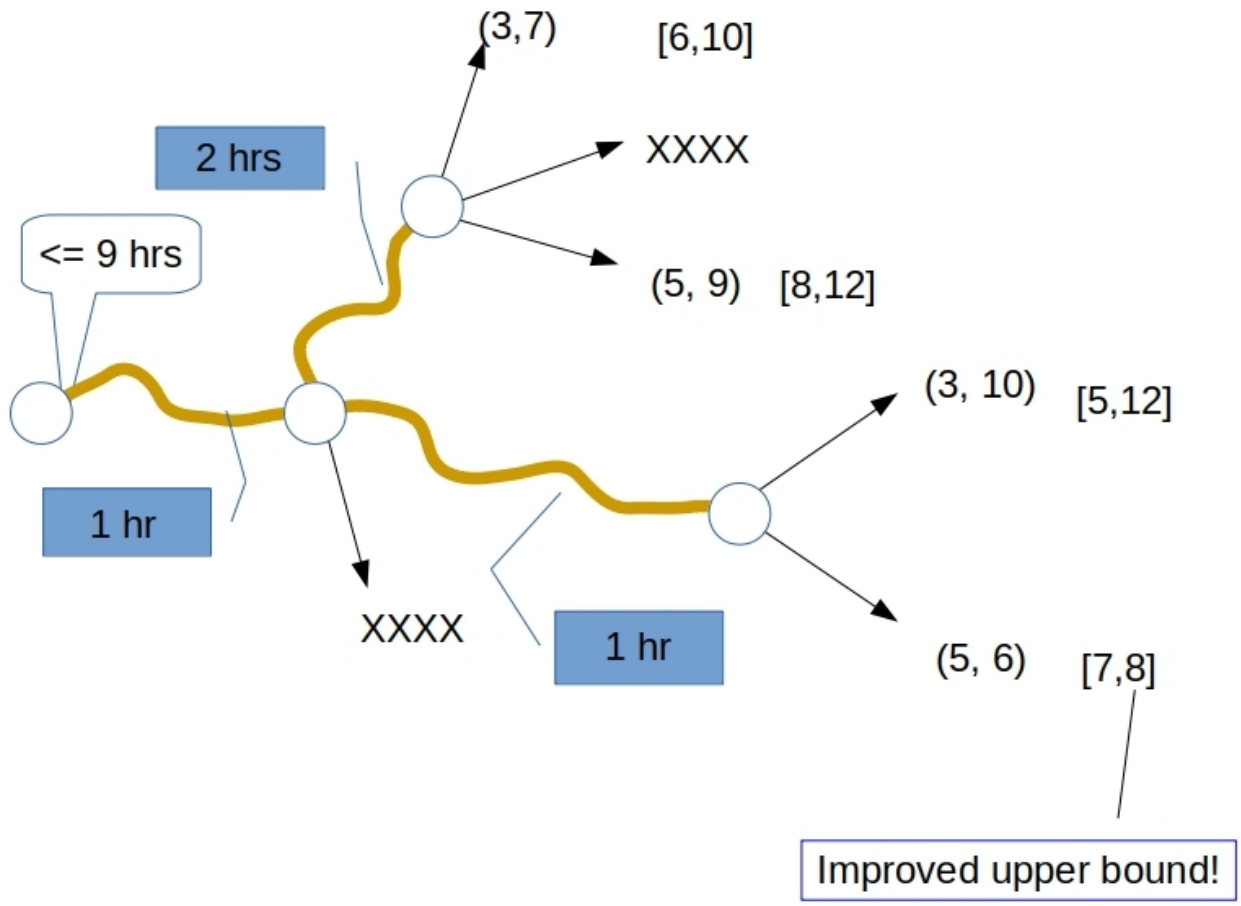


You tell the scout to stay where they are and send another scout down the path labelled [3, 18] because it offers the best possible outcome. This scout travels for two hours down the path and finds another cross-road.



There are three paths from this cross-road, marked as shown in the figure. Once again we can immediately see that one of these cannot lead to the optimal solution so we ignore it. This leaves us with three possible paths, having time brackets of [6, 10], [8, 12] and [5, 11]

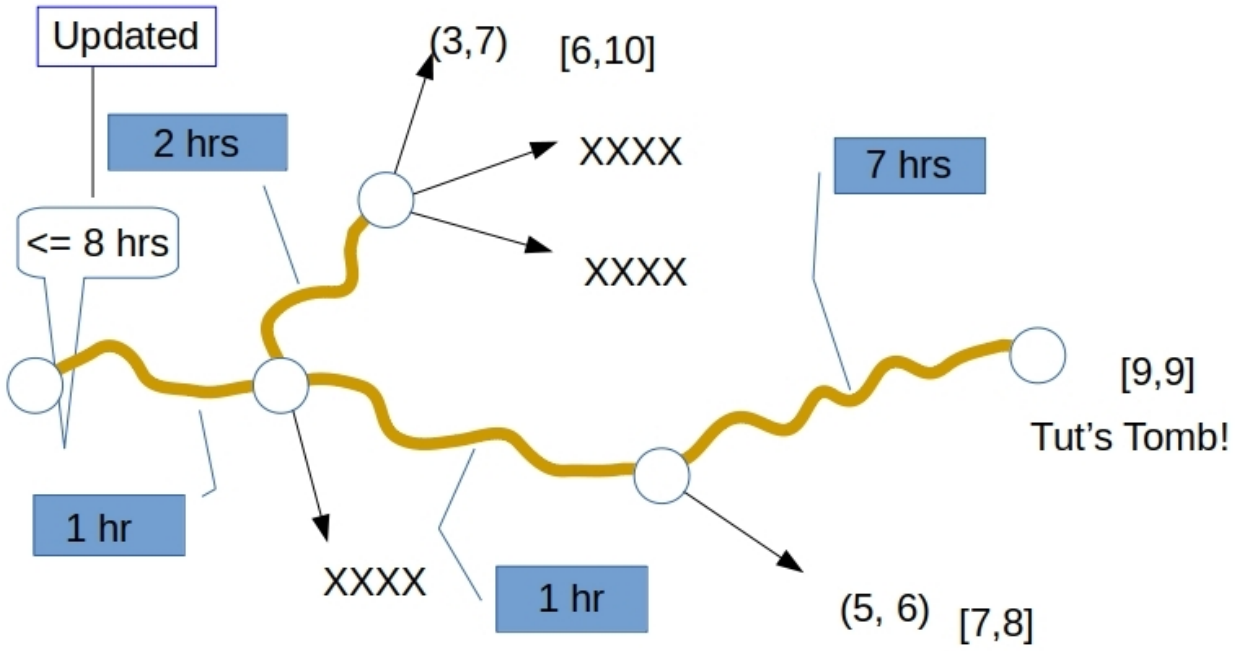
The [5, 11] offers the best possibility so you call the original scout and tell them to go down that path. They walk for one hour and reach another cross-road. This one has only two paths to choose from.



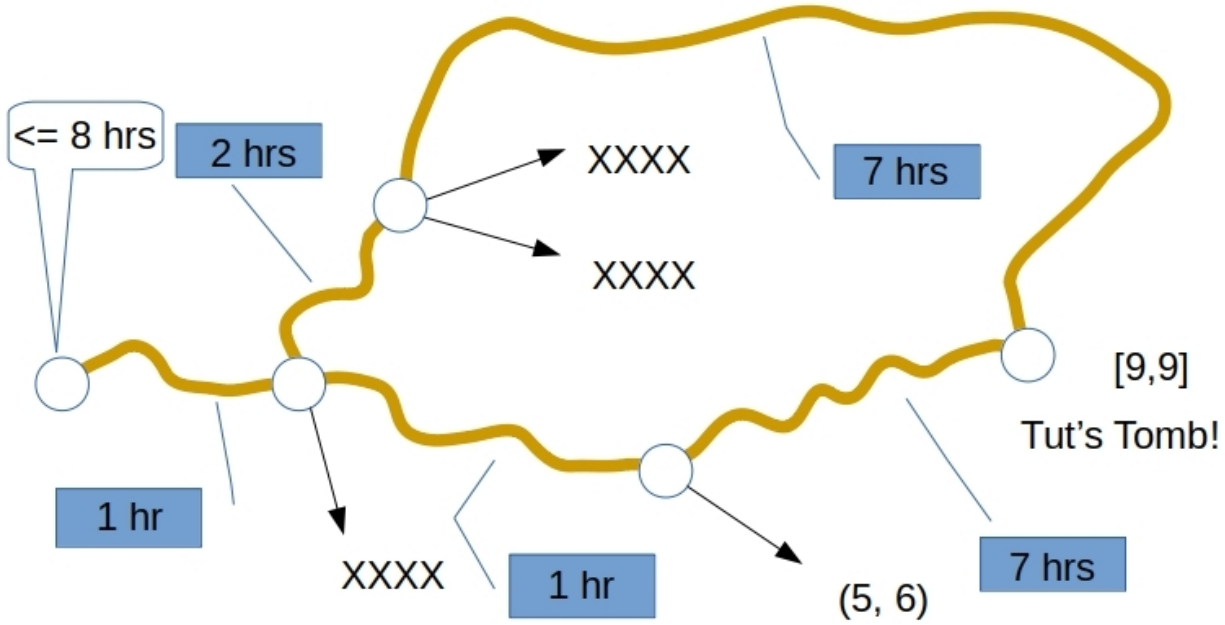
Look! One of the choices has time bracket [7,8] That means if you go that way the total journey will take no more than 8 hours. This is a better upper bound for the journey than you had before: you were originally told the trip would take \le 9 hours – now you know you can do it in \le 8. The significance of this is that one of the other unexplored routes has time bracket [8, 12]. You can now eliminate that path because it can't beat the one you just found.

Of the paths still in the game, the one with time bracket [5, 12] looks the best. You send a scout down that path. They travel 7 hours and arrive at Tut's tomb!!!

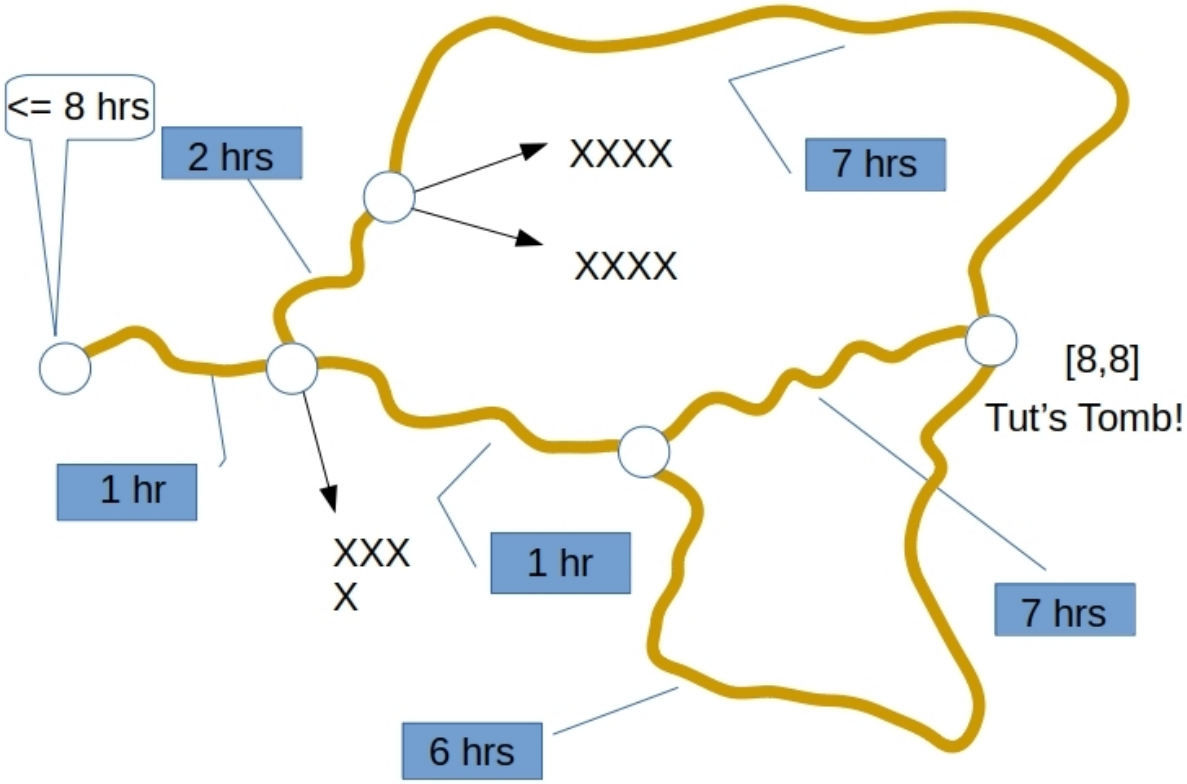
Are you done? You are not! Your scout has arrived at Tut's tomb, but the total travel time along this path is 1+1+7 = 9 .... and you now know you can get there along some other path at a cost of ≤ 8. So you keep giving instructions to the scouts.



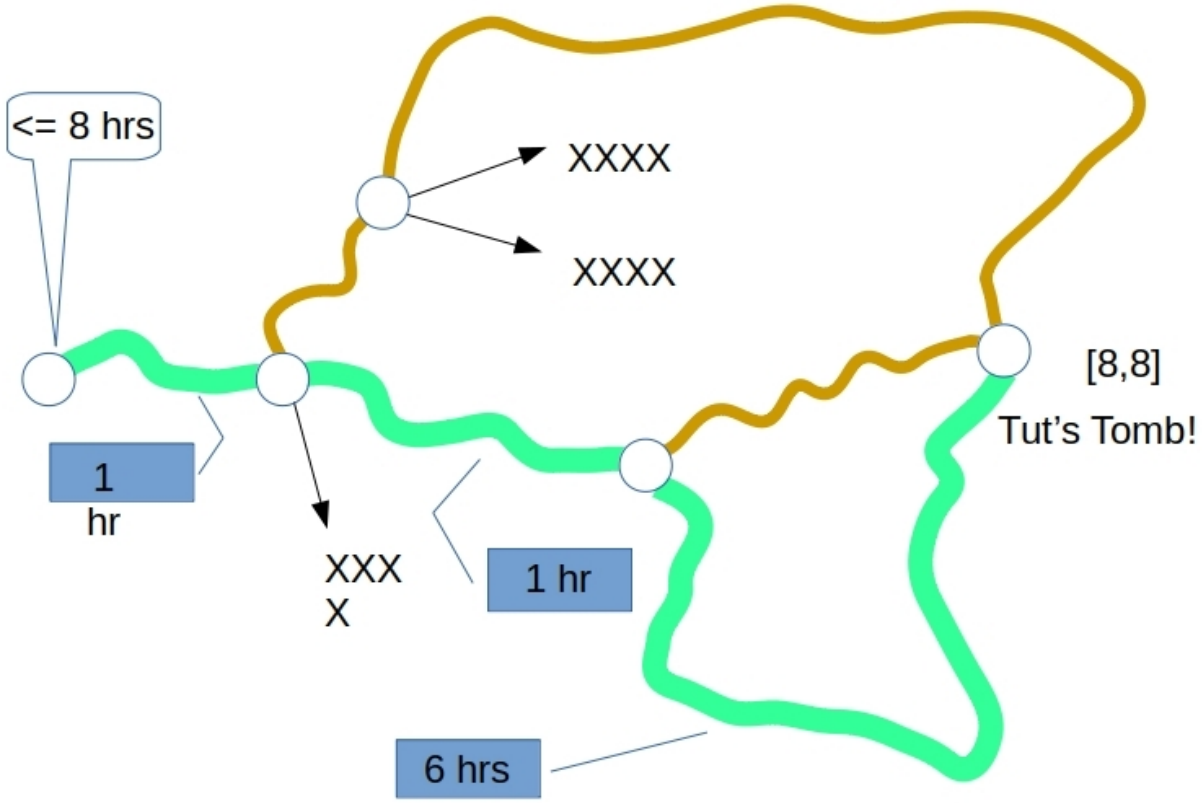
There are only two unexplored paths remaining (remember we eliminated three paths because the lower values in their time brackets were too high). Of the two remaining paths the one with time bracket [6, 10] is most optimistic so you send a scout down that path. After 7 hours they arrive at Tut's tomb. This path has taken a total of 1+2+7 = 10 hours. This is not the road you are looking for.



You are down to just one unexplored path so yo u send a scout down that route. Of course it is possible that the scout might come to yet another cross-road, but mercifully they don't! They arrive at Tut's tomb after walking for 6 hours.



This is the path you were looking for – it has a total travel time of 1+1+6 = 8 hours. We know it is optimal because we have explored or eliminated all the alternatives.



So what's it all about? What are the principles at work here, and why are they useful?

In a Branch and Bound algorithm we develop a set of partial solutions, each with a bracket on the cost of extending the partial solution to a full solution. On each iteration we pick one partial solution and consider all the possible ways to extend it one step further. For each of these new partial solutions we establish a cost bracket. **Some of the new partial solutions may have brackets that allow us to eliminate them immediately**. **Sometimes the new partial solutions let us eliminate other partial solutions that were still under consideration.** We continue until we have explored or eliminated all partial solutions. The best complete solution we encountered is the optimal solution to the problem.

The two emphasized sentences in the previous paragraph give the core justification for using this algorithmic paradigm. Branch and Bound algorithms are used when we can't find a Divide-and-Conquer, Greedy or Dynamic Programming algorithm, or any other clever solution to improve on the BFI method of examining all possible solutions and choosing the best one. Such problems often have O(2^n) possible solutions – enumerating them all is infeasible. But the B&B approach lets us eliminate partial solutions ... which also eliminates all full solutions that are expansions of the partial solutions that we eliminates. If our

algorithm is well designed we may be able to eliminate huge numbers of potential solutions this way – making the algorithm much faster than the raw BFI method. The goal of developing a good B&B algorithm is to develop methods of computing the brackets for partial solutions that

- don't take too long to compute
- give small brackets

As we go forward with our study we will see why each of these is important.

The B&B approach is suited to problems where

- the solution can be described as a sequence of decisions. This often involves choosing a subset of a set. Many of the problems discussed in CMPE/CISC-365 are of this type.
- there is a feasibility constraint that must be obeyed by all solutions
- there is an objective function to be optimized. This often takes the form of finding a solution that minimizes the cost of the elements in the chosen subset, or finding a solution that maximizes the value of the elements in the chosen subset

Without loss of generality, we will assume that our goal is to find a solution with minimum cost.

The B&B algorithmic template looks like this:

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 $[l_P, u_P]$

At this point it is essential to clarify what this bracket represents. The partial solution P can be extended to many full solutions (ie complete decision sequences). Out of all of those full solutions that extend P, let $S*$ be the one with lowest cost. That is, $S*$ is the **best** solution that can be constructed by extending P. Then l_P and u_P must be lower and upper bounds, respectively, on the cost of $S*$. Some important points:

- l_P and u_P **do not** have to bracket the costs of **all** extensions of P ... just the best extension of P
- l_P and u_P do not have to be close to the actual cost of S* (though the algorithm is more efficient if they are). All that absolutely must be true is $l_P \leq (\text{ cost of } S*) \leq u_P$
- we need to compute l_P and u_P without (in almost all cases) knowing what $S*$ looks like

As we construct $\mathcal{P}$, we eliminate any partial solution P with $l_P > U_{Global}$

Now continue to build, expand and eliminate partial solutions until an optimal solution is found:

```
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
```

There is quite a lot to think about in this algorithm (including understanding how we can be sure it returns an optimal solution!) but that will be in the next set of notes.