The Branch and Bound Algorithm Design Technique

Now we want to study a powerful technique that is a combination of brute force—creating a tree of possibilities—combined with clever ways to determine that some nodes of the tree are "non-promising" and need not be pursued further.

As we will see, this technique can sometimes be useful when the best known algorithm is the brute force algorithm, but by "pruning" non-promising branches, we can sometimes find solutions to instances that we could not expect to solve in a reasonable amount of time by the brute force method. This material has a strong "heuristic" flavor—we try to do things that are likely to solve a given instance, but we must always realize that there is a possibility that the algorithm will run, for some instances, for as long as the brute force method.

As an overview, note that optimization problems—and other problems, too—involve decisions. For many problems involving decisions, we can view the brute force method as constructing the tree of all possible decisions. Then the heuristic element of the "branch and bound" technique means that we can often determine that there is no point in creating the subtree rooted at a certain node because we can see that the decisions made up to that point prevent us from doing better than some known result, no matter what decisions we make going on from that node.

To design an algorithm using the possibilities tree (that's the "branch" part of the technique), we need to clearly specify our notation—what decision is being made at each level of the tree, and what the various branches mean. Sometimes we must label the branches, and other times we can use a convention and avoid that work. Also, it is important to store the appropriate information in each node, and to clearly say what that information means. Ideally, we should put enough information in the nodes that we don't have to label the branches, or even literally build the tree.

We can create an algorithm based on a possibilities tree, with or without bounding, in various ways. We sometimes use recursive calls, where the tree is not actually constructed, each node is simply a recursive call, and the necessary information is passed as arguments to the call. Or, we can literally construct the nodes, as objects that store the necessary information in instance variables of the node class.

> In the last week of the course, we will look very briefly at the area known as *computational complexity*. In this area, we study how fundamentally hard a given problem is, rather than studying the efficiency of algorithms that solve that problem. In this study we will see a number of problems—including the 0-1 knapsack problem and the Euclidean traveling salesperson problem—that are, as far as any human knows, extremely difficult to solve, in the sense to be precisely described in that last chapter.
>
> The point here is that this chapter provides one way to tackle hard problems for which no efficient algorithm is known by a branch and bound technique that is not known to solve a given instance in a reasonable amount of time, but on average seems to do well.
>
> In the next chapter, we will study the idea of only *approximately* solving a hard problem in a reasonable amount of time.

Branch and Bound Applied to the 0-1 Knapsack Problem

Let's apply these ideas to the 0-1 knapsack problem.

We will store this information in each node:

> The node number, where nodes are numbered consecutively in order of creation.
>
> The items that have been chosen and rejected at this point (rejected items are shown crossed out).
>
> The profit achieved by the chosen items.
>
> The total weight of the chosen items.
>
> A cleverly computed bound that says no matter how we accept or reject the remaining items, we can't get a better profit than this bound.

Let's explore the branch and bound technique for this problem with this sample instance:

> Let $W = 16$ and use this item data, where we have added the profit per weight data as an aid to computing bounds, and sorted the items according to it:

i	p_i	w_i	$\frac{p_i}{w_i}$
1	40	2	20
2	30	5	6
3	50	10	5
4	10	5	2

We will use a *best first* approach to building the possibilities tree, where the node with the best bound is explored next. Also, we will monitor the best actual profit in any node seen so far, and prune any node whose bound is worse than that profit. We will also, of course, prune any node whose total weight exceeds the allowed capacity.

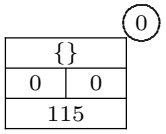
The computation of the bound is the only even slightly difficult part of this algorithm—everything else is quite simple and natural. The bound for a node is computed by saying "what profit could we achieve, by choosing or rejecting any desired *fractional part* of the remaining items?" We also sort the items in order of decreasing profit per weight, which implies that we should use all of item $m + 1$ that fits, until we've used it all, and then use whatever part of item $m + 2$ that fits, and so on.

We will consider a node non-promising if its total weight exceeds W, of course.

⇒ Let's work through the example.

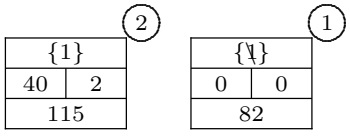
Instead of literally drawing the tree, we will add and remove nodes to and from a *priority queue*. We won't detail the efficient heap data structure based way to implement a priority queue, but will instead add nodes where they belong in sorted order according to bounds.

We start by generating the root node, which is added to the priority queue:



This is node 0, with no items chosen (or rejected), for a profit of 0, and a weight of 0. To figure the bound, we add items—or a fraction of the last item used—in order of profit per weight, until we reach of weight of 16 (or run out of items). So, all of item 1 fits, giving profit of 40 and weight 2. Then all of item 2 fits, for a profit of $40 + 30 = 70$, and a weight of $2 + 5 = 7$. Then only 9 units of item 3 fit, for a weight of 16, and a profit of $70 + 9 \cdot 5 = 115$.

Now we remove (and process) from the priority queue the item with the best bound, which is of course node 0 (since it is the only node in the priority queue), and add its children, obtained by rejecting item 1 (node 1) and accepting item 1 (node 2), obtaining this priority queue:

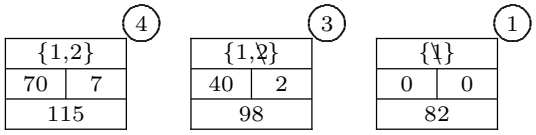


Note that we are arbitrarily creating the left child—the one that rejects the next item—first, and then the right child—the one that accepts the next item.

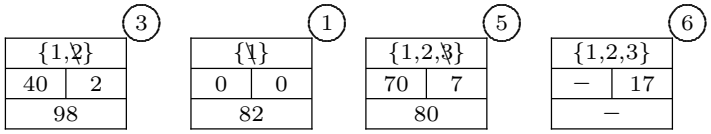
Also note that the nodes in the priority queue can be listed in any order, because we will simulate the behavior of a priority queue by just looking over all the items and picking the one with the best bound.

For node 1 the bound was computed by using all of item 2, for a profit of 30 and a weight of 5, then using all of item 3 for a total profit of 80 and a total weight of 15, and then using 1 weight unit of item 4, which brings the total profit up to 82 and the total weight up to 16.

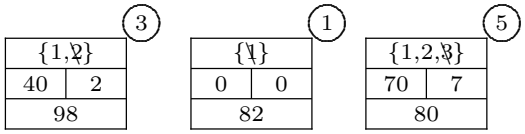
Next, the algorithm removes node 2, because its bound (115) beats the bound 82 of node 1, and adds nodes 3 and 4, yielding (in sorted order, just for mild convenience)



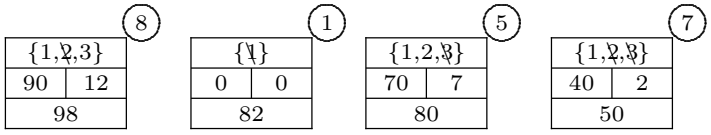
Next we remove node 4 and add its children, nodes 5 (reject item 3) and 6 (accept item 3), giving:



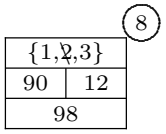
Then node 6 is pruned, because it is too heavy (we probably wouldn't even put it in the priority queue in the first place, but we did here to show it so we could see why it was pruned), giving:



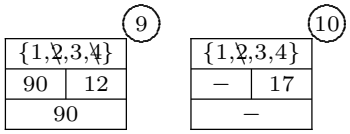
Next node 3 is explored (removed), and nodes 7 and 8 are added, giving:



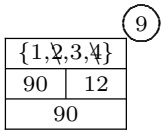
Now something exciting happens: since node 8 actually achieves a profit of 90, we prune all the nodes that have a bound less than or equal to 90, giving:



Next we remove node 8 and add nodes 9 and 10, giving:



Then we prune node 10, since it is too heavy, giving:



At this point we stop and declare node 9 the best, realizing that there are no more items to consider adding.

The Tree Is Usually Implicit

As mentioned earlier, note that the possibilities tree used in thinking about branch and bound algorithms is mostly a conceptual tool. When it comes down to implementing such an algorithm, the nodes can turn into method calls, with the necessary information for each node passed in as arguments to the method calls, or can be stored as separate nodes with no actual tree structure. Partly this is helpful because it is easier to not have to literally build a tree, but mostly it is to save memory space. For example, with our branch and bound 0-1 knapsack algorithm, only the nodes that have been created but not yet

explored have to be stored at any given time. In the worst case this could still be $\Theta(2^n)$ nodes, but we can reasonably hope that far fewer nodes ever need to be stored at one time.

Efficiency Analyses of Two Approaches to 0-1 Knapsack Problem

The basic (fill in the whole chart) dynamic programming algorithm for the 0-1 knapsack problem takes $O(nW)$ time, since the table has n rows and W columns. If we do the "only compute the value of a cell if it is needed" version we need at most 2^j cells on row $n - j$, requiring $O(2^n)$ time in the worst-case. Thus, this version takes $O(\min(nW, 2^n))$ time in the worst-case.

The branch and bound method has worst-case efficiency in $O(2^n)$, in the case that the entire possibilities tree has to be built. The heuristic element of this approach is that very often far less work will need to be done, because many nodes will be pruned as non-promising.

Exercise 30 [10 points] (target due date: Monday, November 16)

Perform the branch and bound algorithm for the 0-1 knapsack problem similarly to the previous example, on the instance given below.

For convenience, you probably will want to actually draw the tree, noting that at any moment the nodes that have no children are the ones that would be in the priority queue, and the node with the best bound can be found by simply scanning those nodes.

Here's the instance to solve:

$W = 15$ and

i	p_i	w_i	p_i/w_i
1	64	4	16
2	90	6	15
3	91	7	13
4	48	4	12
5	33	3	11
6	10	1	10

---

ex. 30

$W = 15$

i	p_i	w_i	p_i/w_i
1	64	4	16
2	90	6	15
3	91	7	13
4	48	4	12
5	33	3	11
6	10	1	10

$$\text{total weight} = wt + \sum_{j=i+1}^{k-1} w_j$$

$$\text{bound} = \left(p_i + \sum_{j=i+1}^{k-1} p_j\right) + \left(w - \text{tot wt} \cdot \frac{p_k}{w_k}\right)$$

weight $\geq w$

$f(p_i) + W_{wj}\left(\frac{p_i}{w_j}\right)$ = Best actual profit

Greedy Approach

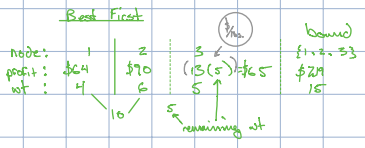
$P_1 \rightarrow P_2 \rightarrow P_4 = f(P_1 + P_2) = 64 + 90 + 48 + 10$

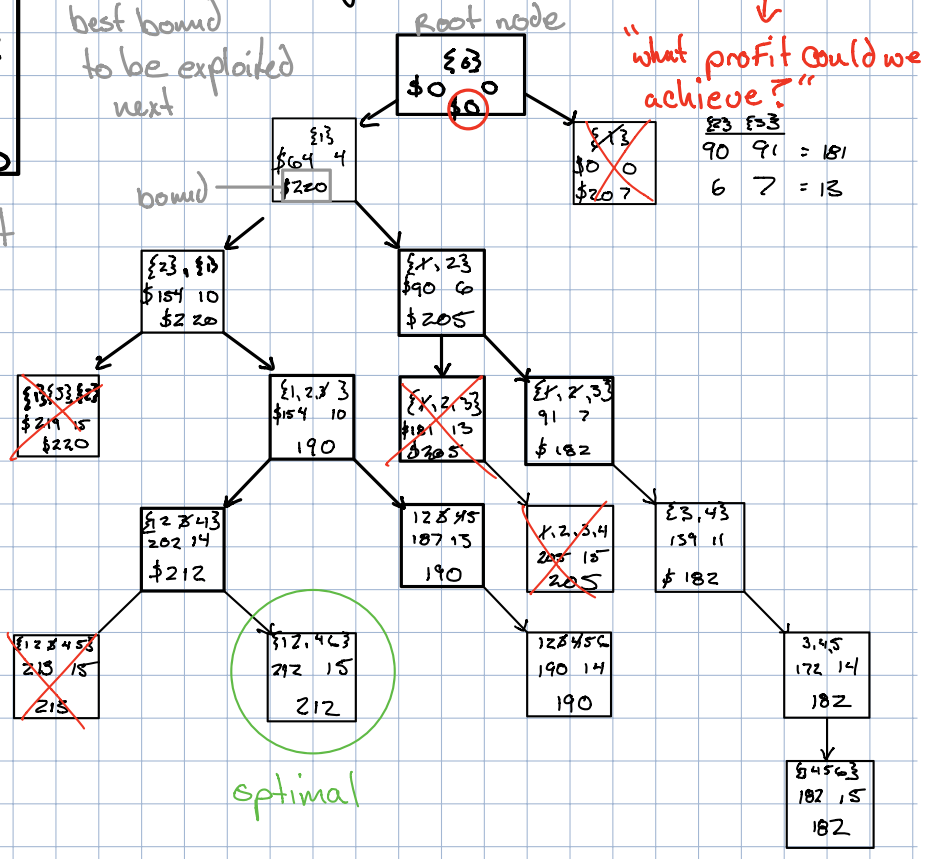
$\rightarrow P_6$

$= \$212$ profit

$w_1 + w_2 = 6 + 4 + 4 + 1$

$= 15$ lbs





x 2	3	{2,3}	{2,3,4}
$90 6	$91 7	$181 13	$205 15

$z(12) + 181$

Exercise 31 [4 points] (target due date: Monday, November 16)

Your job on this Exercise is to write code in Java to implement the branch and bound algorithm for the 0-1 knapsack problem.

Create a Java application named `Ex31` that will ask the user for the name of a data file, read the information from the data file, and then solve that instance of the 0-1 knapsack problem using the branch and bound method, following exactly the algorithm demonstrated in the previous example—without an explicit tree.

The data file must contain the capacity of the knapsack, followed by the number of items, followed by the profit and weight information (on one line) for each item. All these values are integers. You may assume that the items are sorted in order from most profitable per unit of weight to least profitable.

As a small but important requirement to avoid irritating me, you *must* label the items starting with 1, rather than 0.

Here is a sample data file:

```
12
6
100 4
120 5
88 4
80 4
54 3
80 5
```

Here is a sample run on this data file, showing the output your program must produce (within cosmetic differences), and, implicitly, the algorithm it must use:

```
Capacity of knapsack is 12
Items are:
1: 100 4
2: 120 5
3: 88 4
4: 80 4
5: 54 3
6: 80 5

Begin exploration of the possibilities tree:

Exploring <Node 1:   items: [] level: 0 profit: 0 weight: 0 bound: 286.0>
      Left child is <Node 2:   items: [] level: 1 profit: 0 weight: 0 bound: 268.0>
         explore further
      Right child is <Node 3:   items: [1] level: 1 profit: 100 weight: 4 bound: 286.0>
         explore further
         note achievable profit of 100

Exploring <Node 3:   items: [1] level: 1 profit: 100 weight: 4 bound: 286.0>
      Left child is <Node 4:   items: [1] level: 2 profit: 100 weight: 4 bound: 268.0>
         explore further
      Right child is <Node 5:   items: [1, 2] level: 2 profit: 220 weight: 9 bound: 286.0>
         explore further
         note achievable profit of 220
```

```
Exploring <Node 5:   items: [1, 2] level: 2 profit: 220 weight: 9 bound: 286.0>
      Left child is <Node 6:   items: [1, 2] level: 3 profit: 220 weight: 9 bound: 280.0>
         explore further
      Right child is <Node 7:   items: [1, 2, 3] level: 3 profit: 308 weight: 13 bound: 308.0>
         pruned because too heavy

Exploring <Node 6:   items: [1, 2] level: 3 profit: 220 weight: 9 bound: 280.0>
      Left child is <Node 8:   items: [1, 2] level: 4 profit: 220 weight: 9 bound: 274.0>
         explore further
      Right child is <Node 9:   items: [1, 2, 4] level: 4 profit: 300 weight: 13 bound: 300.0>
         pruned because too heavy

Exploring <Node 8:   items: [1, 2] level: 4 profit: 220 weight: 9 bound: 274.0>
      Left child is <Node 10:   items: [1, 2] level: 5 profit: 220 weight: 9 bound: 268.0>
         explore further
      Right child is <Node 11:   items: [1, 2, 5] level: 5 profit: 274 weight: 12 bound: 274.0>
         hit capacity exactly so don't explore further
         note achievable profit of 274

Exploring <Node 2:   items: [] level: 1 profit: 0 weight: 0 bound: 268.0>
      pruned, don't explore children because bound 268.0 is smaller than known achievable profit 274

Exploring <Node 4:   items: [1] level: 2 profit: 100 weight: 4 bound: 268.0>
      pruned, don't explore children because bound 268.0 is smaller than known achievable profit 274

Exploring <Node 10:   items: [1, 2] level: 5 profit: 220 weight: 9 bound: 268.0>
      pruned, don't explore children because bound 268.0 is smaller than known achievable profit 274

 Best node: <Node 11:   items: [1, 2, 5] level: 5 profit: 274 weight: 12 bound: 274.0>
```

Your application must display on screen each node when it is first reached, showing the items selected at that node, the total profit for those items, the total weight for those items, and the bound on that node.

Note that your algorithm must use a priority queue to replace manually looking at all the nodes in the current tree to determine which has the best bound. Whenever a node is explored, it should either be pruned, or should be removed from the priority queue with both its children added to the priority queue.

You may implement the priority queue without worrying about efficiency (if you use a heap data structure removing the highest priority item and adding an item will both have efficiency in $\Theta(\log n)$, but it is okay to just use a list of some kind and have removing the highest priority—best bound—node take $\Theta(n)$, with adding being in $\Theta(1)$.

Be sure to test your algorithm thoroughly.

To submit your work, send me an email with the single file `ex31.zip` attached, containing all the source code for your project, in portable form (no meaningless `package` statements or hard-coded file names).

Branch and Bound Approach to ETSP

In the last part of Chapter 11, we tried to solve ETSP by viewing an instance of ETSP as an instance of LP and using the simplex method to solve the LP, but we saw that we had no way to enforce the constraint that all the intensities be either 0 or 1—we saw that quite often instances of ETSP led to an instance of LP whose solution was not the optimal tour because it had intensities of 1/2. Also, we saw that to force one connected graph, we had to add cut constraints. In this section, we will see how we can use this previous work to provide a *bound* that will let us solve ETSP heuristically using a branch and bound algorithm.

> With end of the semester looming, I did not, and will not, bother you with an Exercise on formulation of the LP that attempts to be equivalent to ETSP, but make sure that you understand exactly where the LP on page 11.13 comes from, and that you understand the discussion of cut constraints on page 11.15—there will almost certainly be a Test 3 question involving this material.

We can now describe the branch and bound algorithm for solving ETSP.

The idea is very simple:

> when we have an LP, including any cuts (note that all tours satisfy all cuts, so adding a cut constraint will never eliminate any tour from the feasible set, so we can add cuts freely), whose optimal solution includes one or more of these nasty edges with non-integer intensity, corresponding to some variable x_{kj} having a value strictly between 0 and 1 (for example $\frac{1}{2}$), we can *branch* by adding a constraint $x_{kj} = 0$ or $x_{kj} = 1$. Then we can use our usual ideas for branch and bound, eventually, we hope, reaching an LP whose solution is a tour, and eventually one whose solution must be optimal.

First, note that it is easy to add constraints $x_{kj} = 0$ or $x_{kj} = 1$ to an LP, without actually adding stuff.

> After creating a tableau with any desired cuts added as constraints, we can then go through and for each `zero k j` command (for `HeuristicTSP`) simply delete the column for x_{kj}, the column for s_{kj}, and the constraint row $x_{kj} + s_{kj} = 1$. For each `one k j` command, we do those same things, except before deleting the column for x_{kj}, we subtract all its components from the corresponding right hand side vector.
>
> Actually, this is the approach taken by the version of `Tableau` in the `AutoHeuristicTSP` sub-folder—the one in `ETSP` is slightly older (and the newer one is slightly incompatible with `HeuristicTSP`).

Here is a heuristic algorithm, using the branch and bound technique, for solving, perhaps, any given ETSP instance:

> Get the root of the branch and bound tree by solving the original version of the related LP. If the optimal solution gives disconnected sub-tours, add in `cut` constraints until one connected graph is obtained.

If the graph is a tour, it is the optimal tour, and we can stop with the solution to the ETSP.

Otherwise, look for a "red" edge (intensity not 0 or 1). Branch on the choice of that edge being forced to be either present ($x_{kj} = 1$) or not ($x_{kj} = 0$).

Repeat in the usual branch and bound way, pursuing a depth-first search for a tour (to reach a tour more quickly for bounding purposes), or using a best-first search (to pursue the most promising branches first). When a tour is obtained, it is the optimal tour for the original problem plus the specific choices made along the branch to reach it. It therefore provides a bound on all the other nodes—if any node has a worse score than this tour, then that node can be pruned.

For each node reached, we can freely add cut constraints until a single connected graph is obtained. The scores will get worse, because we are adding constraints, but we will never be eliminating tours by adding these constraints.

A Small Example of the Branch and Bound Algorithm for TSP

⇒ Consider the Sorensen problem, specified in a data file creatively named `sorensen`. We apply `HeuristicTSP` to the base problem, obtaining the optimal solution with $z = 177.8145$ (rounding to 4th decimal place, and ignoring the $-$ introduced by doing minimization instead of maximization).

Next we pick a variable, say x_{15}, on which to branch. We add `zero 1 5` to the `sorensen` data file and obtain an optimal solution to the corresponding LP with $z = 180.6394$. Since this solution is a tour, we prune the tree below that node.

Then we consider the other possibility, and replace the $x_{15} = 0$ constraint with $x_{15} = 1$. This gives an optimal solution to the corresponding LP with $z = 177.8145$, which is a tour.

The optimal choice, resulting in the optimal tour, is obviously the node produced by using $x_{15} = 1$.

A Bigger Example of the Branch and Bound Algorithm for TSP

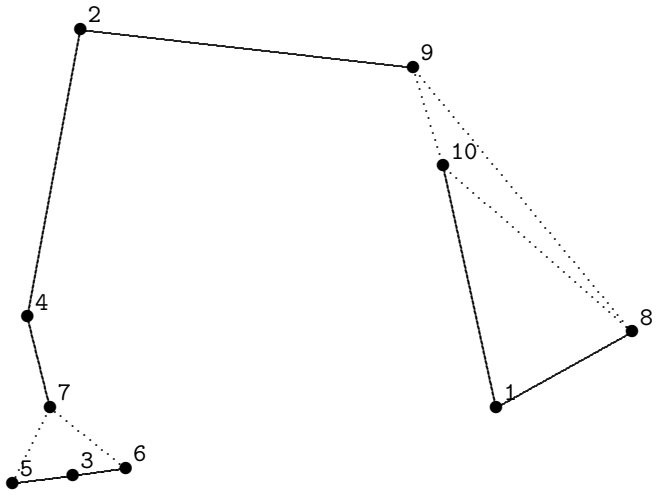
⇒ Let's solve this instance of ETSP with 10 points (instance (in the data file `bigger`)

```
74 17
19 67
18 8
12 29
10 7
25 9
15 17
92 27
63 62
67 49
```

We will make various choices in the following in order to illustrate the branch and bound ideas. Later we will see that we can solve this problem in an easier way by making different choices.

⇒ As we go through this example, we will write on scratch paper a sketch of our branch and bound tree, showing the branches we have made with choices to require some x_{jk} to be either 0 or 1, with each node containing its score, which is the objective function value for the corresponding LP at that node.

If we run `HeuristicTSP` on this problem, we obtain this optimal point for the LP, where solid lines represent decision variable values of 1 (blue edges), and dotted lines represent decision variable values of $\frac{1}{2}$ (red edges):

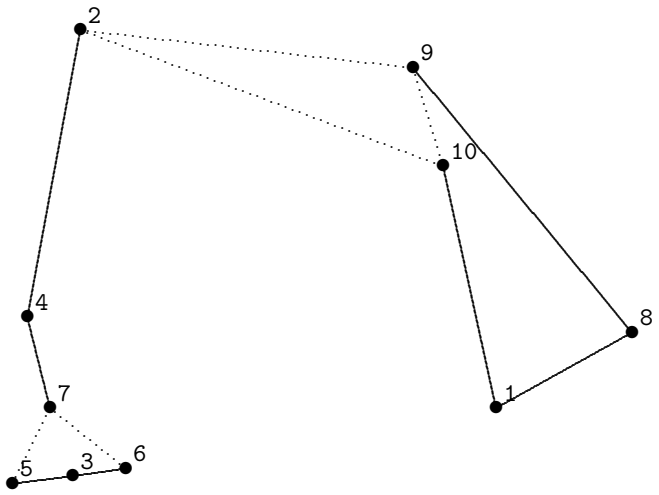


Since the graph is connected, we don't use a cut (we could do `cut 2 3 4 5 6 7` to good effect, but we want to demonstrate the branch and bound algorithm, so we won't), but instead pick a red/dotted edge on which to branch. Let's use $x_{8,9}$, because it looks like it might or might not want to be in the optimal tour (picking a good edge on which to branch is a heuristic element of this algorithm).

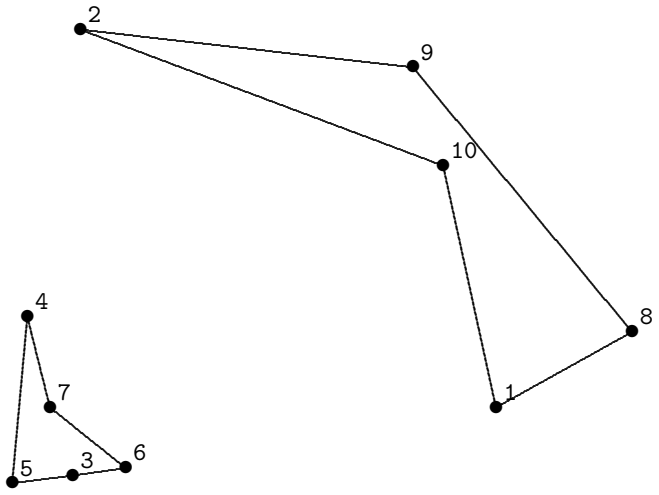
So, we add

`one 8 9`

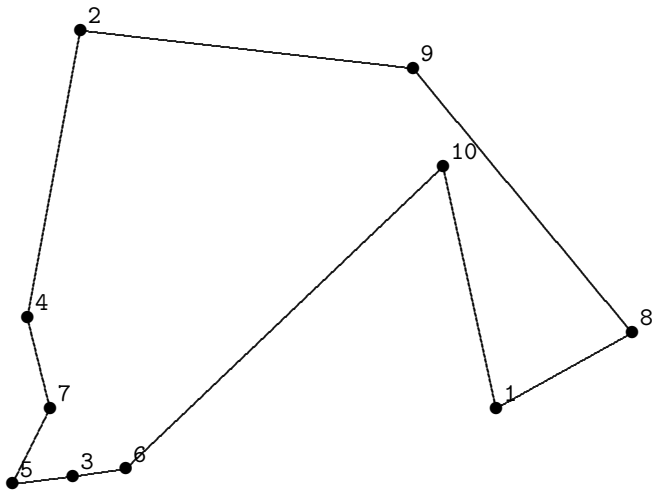
to our data file and run the application again, obtaining this graph:



Again we have a connected graph with some red/dotted edges, so we branch on x_{29}. If we use $x_{29} = 1$, we obtain

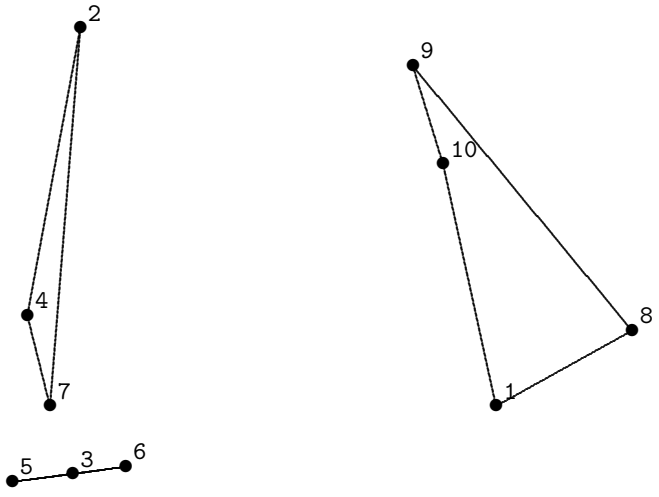


Now we add `cut 3 4 5 6 7` to try to eliminate the two sub-tours, and we obtain

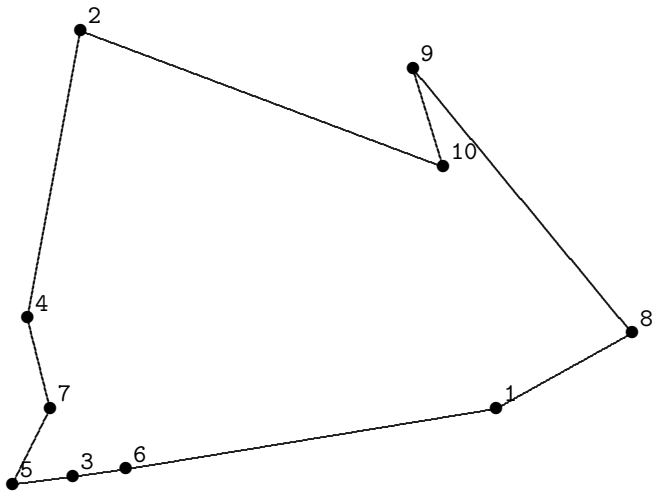


with a score (total weight) of 278.40673. We are now very excited because we've found a tour, which means that if we find any node with score bigger than 278.40673, we can prune it.

Now we have one unresolved node, so we use the branching choices $x_{8,9} = 1$ and $x_{2,9} = 0$ and obtain

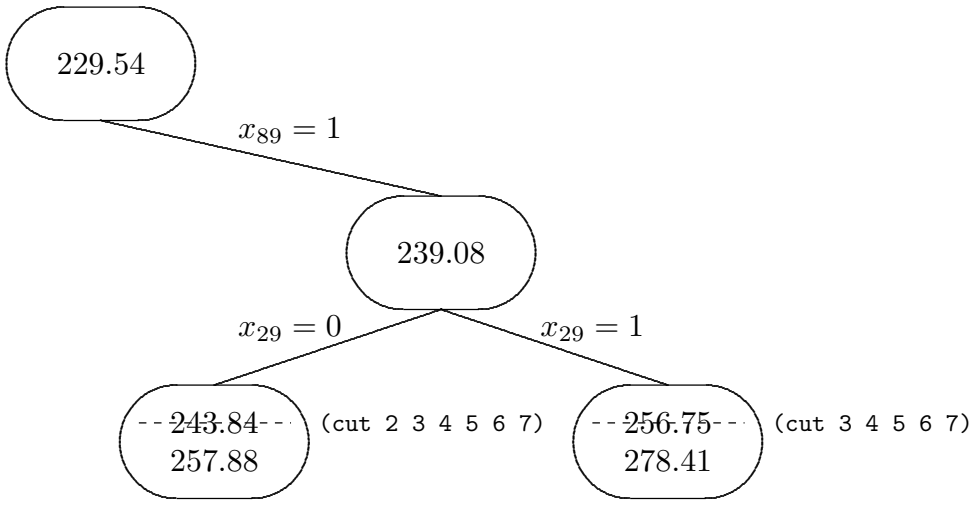


with a score of 243.83717. Since this score is better than our bound, we can't prune this node. But, since it is disconnected, we can add a cut, say `cut 2 3 4 5 6 7` yielding

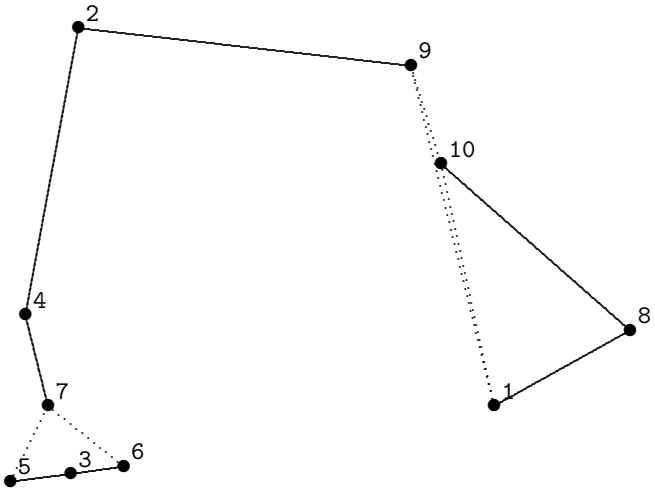


with a score of 257.88113. Now we stop, since it's a tour, and notice that we have found a tour with a better score than our previous best (only, actually) tour, so we update our bound and continue.

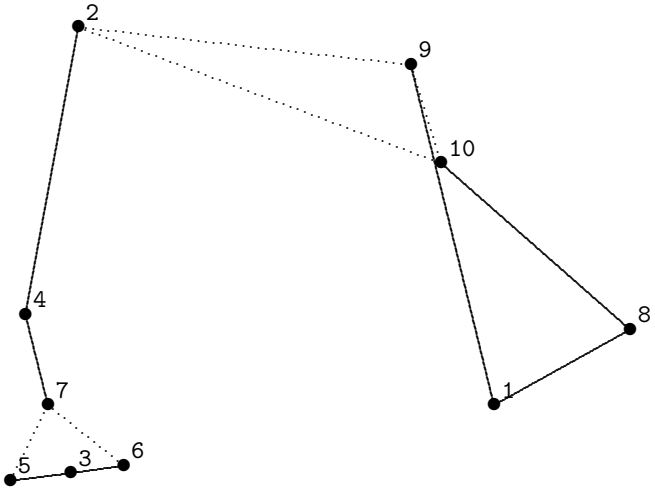
At this point our hand-drawn sketch of the branch and bound tree looks like this:



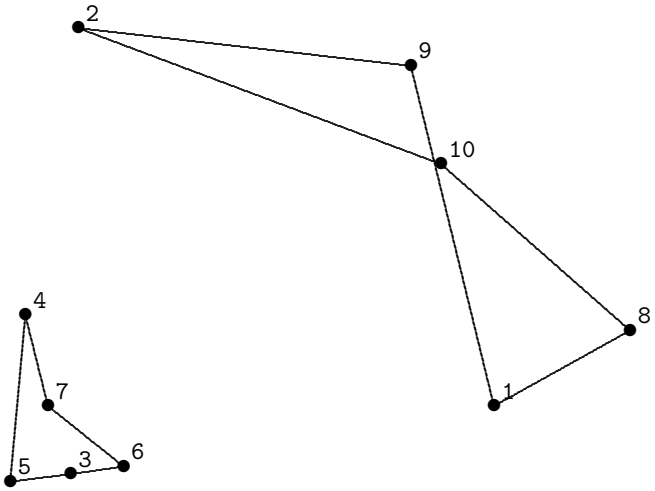
Now we pursue the possibility, back at the root node, that x_{89} should be 0, obtaining



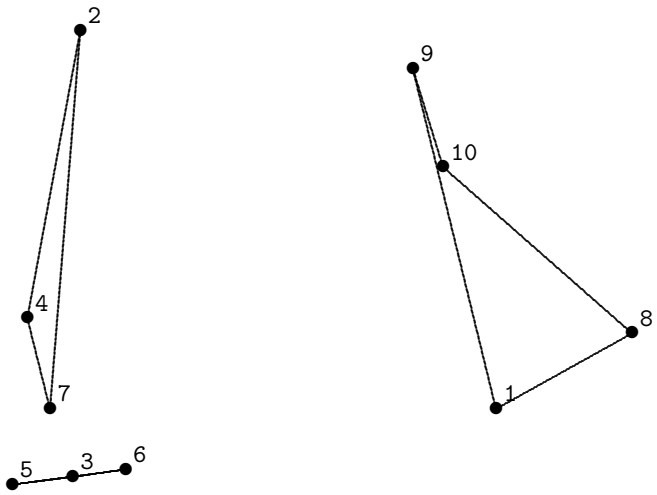
with a score of 230.21930. We decide to branch from here on $x_{1,9}$. If we pursue $x_{1,9} = 1$, we obtain



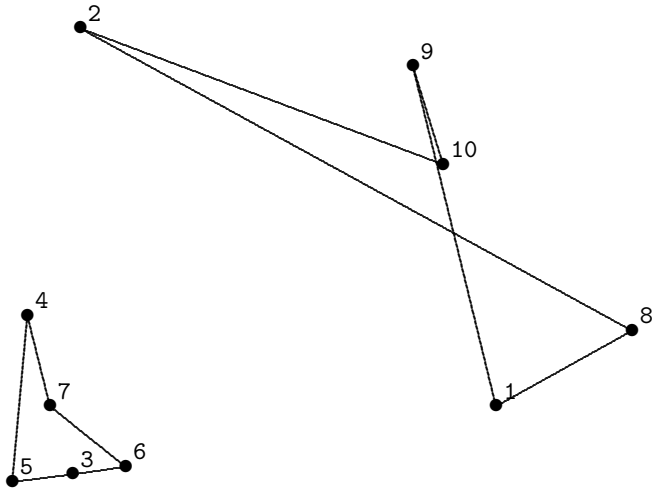
Now we branch from here, first using $x_{2,9} = 1$ and obtaining



with a score of 258.16466. Since this is worse than our current best known tour score (257.88), we prune this node (since a cut will only make its objective function value worse, we don't bother). On the other branch, with $x_{2,9} = 0$, we obtain

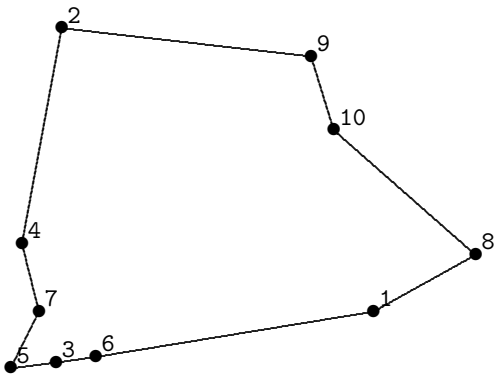


with a score of 245.25381. This means, sadly, that we should pursue this node further. Since it is disconnected, we'll do `cut 2 3 4 5 6 7`, and obtain



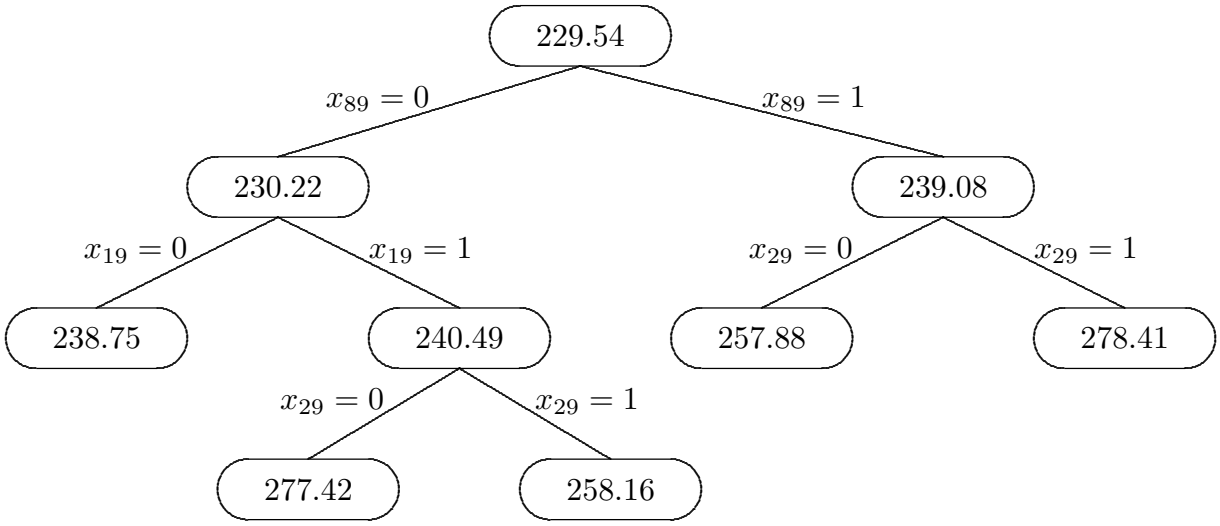
with a score of 277.42192, which allows us, thankfully, to prune this node.

Now we go back to the only unresolved node, with $x_{8,9} = 0$ and $x_{1,9} = 0$, obtaining



with a score of 238.74866. This is, obviously, the optimal tour, and we are done!

Here is the possibilities tree that we created through this process (with objective function values rounded to two decimal places):



Note that if we had carefully pursued nodes with better scores first, we would have found this much sooner (but I've rigged the choices in this example to show more action). This is a heuristic sort of thing, using the node with better score first, even though the scores don't mean anything provable unless the corresponding graph is a tour.

Also, it turns out that we can profitably make cuts when a whole "peninsula" of vertices is only connected to the others by two red edges.

This works because such a cut constraint says that all the edges crossing from the cut points to the other points have to sum to at least 2, which forces more connections than the total of 1 resulting from the 2 red edges at 1/2 each. This seemed to save a lot of branching compared to tackling red edges by branching on them.

Semi-Automated Branch and Bound Algorithm for ETSP

⇒ Let's solve the previous instance of ETSP `bigger` using `AutoHeuristicTSP`, by way of demonstrating how this program can be used.

In the folder `AutoHeuristicTSP` you will find the application `AutoHeuristicTSP` that somewhat automates doing the best-first version of the branch and bound algorithm. It eliminates the need to manually edit the input file by providing these handy commands:

Key	Mnemonic	Action
c	cut	enter a sequence of vertex numbers separated by spaces and terminated by `enter`, and can hit `backspace` as desired, specifying a cut constraint to tbe added
b	branch	enter a pair of vertex numbers specifying a desired branch—nodes setting that variable to both zero and one are added to the priority queue
t	tour	inform the application that the current node gives a tour so it can update its knowledge of the best tour found so far

After solving each LP instance corresponding to a node in the branch and bound tree, the result is shown graphically, with intensity 1 edges shown in blue, intensity 0 edges not drawn, and intensity $\frac{1}{2}$ edges shown in red (edges can have other intensities—they are set to different, other colors). Edges that have been set to 0 in the current node will be shown in white, and edges that have been set to 1 will be shown in black.

Also, as you can see by examining the output in the console window, this application maintains a priority queue of unexplored nodes and always explores the node with the best score, putting it on display so you can cut it repeatedly, and then branch as needed.

You might find it interesting to explore the program `AutoTSPtext`. It is a non-graphical version of `AutoHeuristicTSP`, including helpful reports that we can infer from staring at the picture. This work suggests that it wouldn't be that hard to fully automate this branch and bound algorithm for ETSP—we would just have to have some way of deciding how to branch.

Exercise 32 [10 points] (target due date: Monday, November 16)

Your job on this Exercise is to solve some fairly large ($n = 30$) instances of ETSP using `AutoHeuristicTSP`.

> For an instance of ETSP with $n = 30$, the dynamic programming approach would take $\Theta(n^2 2^{n-1})$ or so steps, which is some multiple of 900 billion steps, requiring a table with roughly a billion rows and 30 columns).
>
> So, it should be fairly impressive that our new approach will be able to solve such problems quite easily!

To do this, just run `AutoHeuristicTSP` with the desired data file as input, and interactively do cuts and branches until the optimal tour is found.

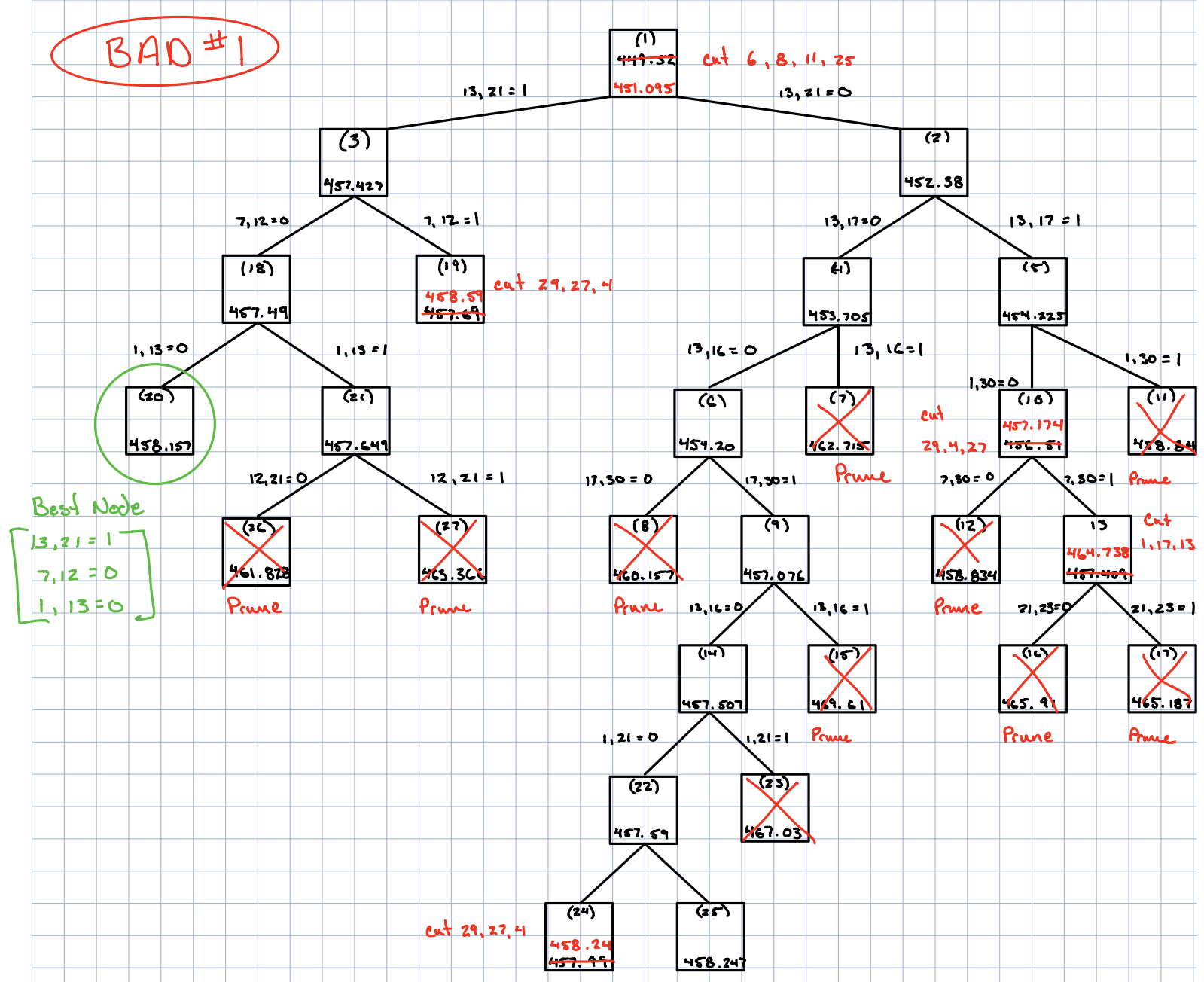
First solve the instance `bad1`, drawing by hand some (until you are sure you understand how to draw the entire tree, and get tired) of the branch and bound tree following the style used on page 9.16 (`AutoHeuristicTSP` keeps track of everything for you, so you don't really need to draw the tree to solve the instance, but I'm requiring it to force you to practice what you'll need for Test 3).

Then, solve the instance `bad2`.

For both instances, report the optimal tour total length and the branching choices that led to the node for this optimal tour in the branch and bound tree.

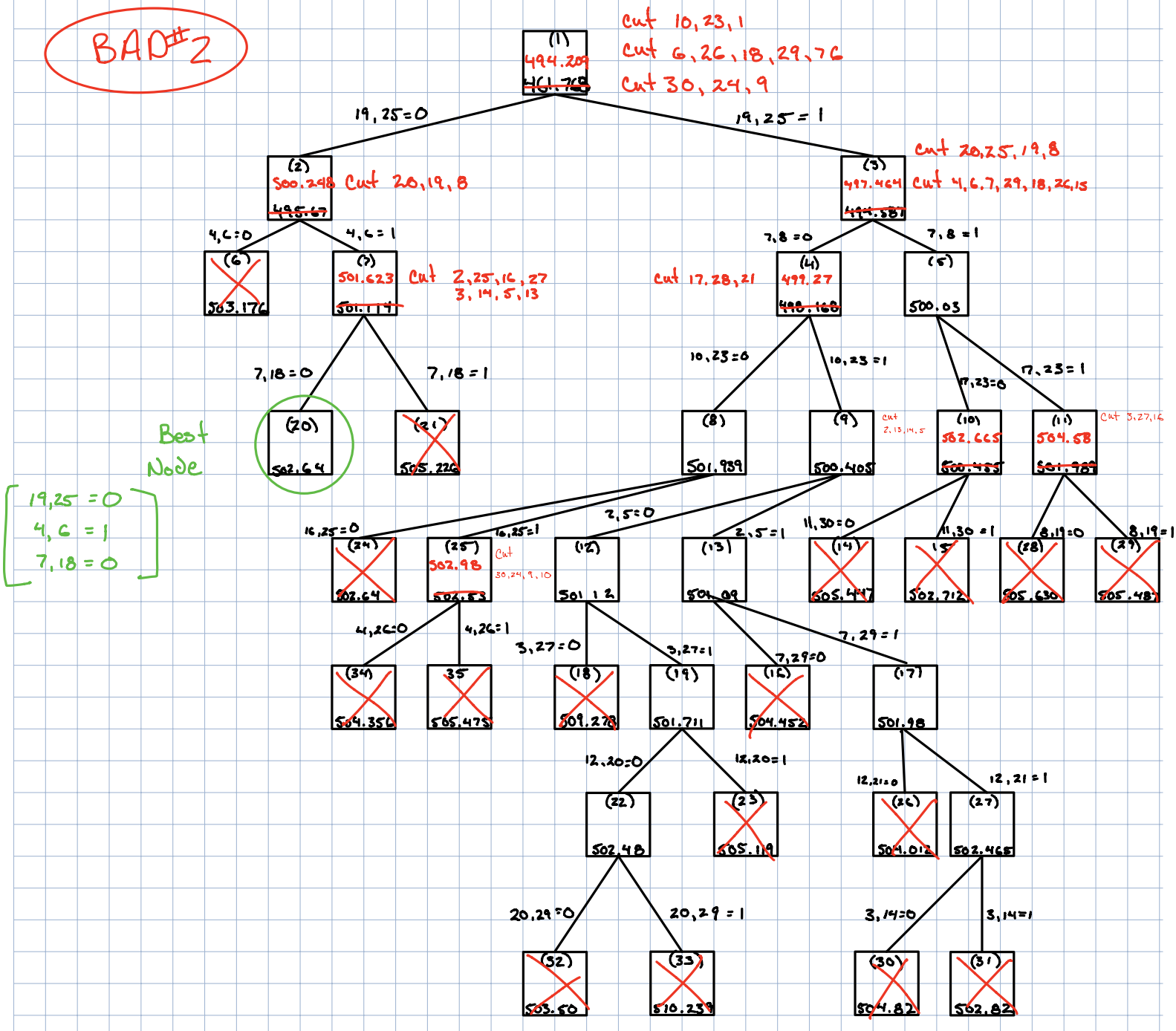
> I came up with `bad1` and `bad2` by using `RandomTestingHeuristicTSP` with $n = 30$ to generate random test instances with 30 points and picking ones that looked hard.

BAD #1



Best Node

[13, 21 = 1
7, 12 = 0
1, 13 = 0]


BAD #2
(1)
494.209
491.768
Cut 10,23,1
Cut 6,26,18,29,7C
Cut 30,24,9
19,25=0
19,25=1
(2)
500.298
Cut 20,19,8
(3)
497.464
Cut 4,6,7,29,18,26,15
Cut 20,25,19,8
4,C=0
4,C=1
(6)
503.176
(7)
501.623
Cut 2,25,16,27
3,14,5,13
7,8=0
7,8=1
(4)
499.27
Cut 17,28,21
(5)
500.03
7,18=0
7,18=1
(20)
502.64
(21)
505.226
Best Node
[19,25=0
4,6=1
7,18=0]
10,23=0
10,23=1
17,23=0
17,23=1
(8)
501.989
(9)
500.405
Cut 2,13,14,5
(10)
502.665
(11)
504.58
Cut 3,27,16
16,25=0
16,25=1
2,5=0
2,5=1
11,30=0
11,30=1
8,19=0
8,19=1
(24)
502.64
(25)
502.98
Cut 30,24,9,10
(12)
501.12
(13)
501.09
(14)
505.447
(15)
502.712
(28)
505.630
(29)
505.487
4,26=0
4,26=1
3,27=0
3,27=1
7,29=0
7,29=1
(34)
504.350
35
505.475
(18)
509.278
(19)
501.711
(16)
504.452
(17)
501.98
12,20=0
12,20=1
12,21=0
12,21=1
(22)
502.48
(23)
505.119
(26)
504.012
(27)
502.465
20,29=0
20,29=1
3,14=0
3,14=1
(32)
503.50
(33)
510.239
(30)
504.82
(31)
502.82